The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated 15 July 2003

For official use only

THE PATENT OFFICE
F
28 JUN 2002
LONDON

01JUL02 E729597-2 D10092
P01/7700 0.00-0215029.0

Your reference **Strand Execution (UK)**

0215029.0

The **Patent Office**

Request for grant of a Patent

28 JUN 2002

Form 1/77

Patents Act 1977

1 Title of invention

Strand Based Execution

2. Applicant's details

[X] First or only applicant

2a If applying as a corporate body: Corporate Name

Critical Blue Ltd

Country

GB

2b If applying as an individual or partnership

Surname

Forenames

2c Address

The Scottish Microelectronics Centre
The Kings Buildings
West Mains Road
Edinburgh

UK Postcode EH9 3JF

Country GB

ADP Number 8413775001

☐	Second applicant (if any)	
2d	Corporate Name	
	Country	
2e	Surname	
	Forenames	
2f	Address	
	UK Postcode	
	Country	
	ADP Number	
3	Address for service	
	Agent's Name	Origin Limited
	Agent's Address	52 Muswell Hill Road London
	Agent's postcode	N10 3JR
	Agent's ADP Number	C03274 7270457002

4 Reference Number

Strand Execution (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐ No ☒

Number of earlier application or patent number

Filing date

15 (4) (Divisional)	8(3)	12(6)	37(4)
☐	☐	☐	☐

6 Declaration of priority

Country of filing	Priority Application Number	Filing Date

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes [ ] No [X]

8 Checklist

		Continuation sheets	
Claims	0	Description	40
Abstract	0	Drawings	0

Priority Documents	~~Yes~~/No
Translations of Priority Documents	~~Yes~~/No
Patents Form 7/77	~~Yes~~/No
Patents Form 9/77	~~Yes~~/No
Patents Form 10/77	~~Yes~~/No

9 Request

We request the grant of a patent on the basis of this application

Signed: Origin Limited Date: 28 June 2002

(Origin Limited)

Strand Based Execution

1 Problem Statement

Very Long Instruction Word (VLIW) architectures are able to perform a large number of parallel operations on each clock cycle. However, the characteristic of most non-numerical code is that there are a large number of dependencies between instructions. That is, one instruction is reliant upon the results of a previous instruction and so cannot be executed concurrently with it. This means that the instruction stream often becomes sparse with many functional units unused during many cycles. This results in an inefficient code representation. Many slots within the instruction word become "No-Operation" (NOP) placeholders because no suitable operation is available for execution.

A further constraint on parallelism is the number of branches that occur within code. In non-numeric applications, a conditional branch operation is generally performed every few instructions. A branch diverges the instruction stream so that different operations are performed depending on the condition. This also restricts the number of parallel operations on a VLIW processor. Branches also cause problems with the operation of the pipelines used in processors. These pipelines fetch instructions several clock cycles before the instructions are actually executed. Thus the architecture needs to know several instructions in advance what the next instruction is going to be. If that is dependent on some conditional that is only calculated just before the branch then it is difficult to avoid a pipeline stall. During a stall the processor performs no useful work for several cycles until the correct instruction is fetched and works its way down the pipeline.

Another significant issue with achieving high levels of code parallelism is the memory alias problem. In a languages such as C or C++ there is heavy usage of pointer memory accesses. It is extremely difficult, and often impossible, to trace data flow within a program to determine the set of objects that a particular pointer might access at any particular time. This imposes severe restrictions on performing load and store operations out-of-order. Whenever a store operation is performed via a pointer it could potentially write to any address. Thus subsequent loads cannot be moved earlier than the store in case they are "aliased" with the store. This severely restricts parallelism since, in most cases, the memory accesses are not actually aliased.

2 Prior Art

A common representation scheme for VLIW architectures is to include some type of NOP compression. This is often based around using a single bit for each instruction field to flag whether it represents a NOP. The instruction word becomes variable in length depending upon the number of NOPs it contains. This tends to complicate the instruction fetch logic.

Most high-end processors include some form of branch prediction scheme. There are many levels of solution complexity, but all try and guess which way a particular branch will go on the basis of compiler analysis and the history of which way the branch has gone in the past. Many of these processors can then speculatively execute code on the assumption the branch will go a particular way. The results from this speculative execution can then be undone (or squashed) should the assumption prove to be incorrect. Some architectures have a predicated execution mechanism. This allows some branches to be simplified by eliminating the branch and executing its target code conditionally. However, this technique can generally only be applied to a limited set of branches in code.

Some high end processors have hardware blocks that analyze the addresses for stores as they are calculated during program execution. They can then be compared against subsequent loads. This allows greater parallelism as loads can be issued earlier than the store. If there is an address match then the hardware takes corrective action, such as re-executing the load after the store is complete. However, such processors are extremely complex and are not suitable for lower cost embedded applications. Some architecture/compiler combinations can generate code to statically issue loads before potentially aliased stores. Additional code is then generated to later compare the addresses and branch to special compensation code to preserve correct program semantics in the unlikely event that the accesses are indeed aliased. Unfortunately this adds significant code size overhead and can only be used in limited cases.

3 Summary of Contribution

CriticalBlue uses a fixed instruction word without NOP compression. This simplifies the instruction fetch/decode stages of the processor. All execution in a CriticalBlue processor occurs within code blocks called regions. A region is simply a fixed sequence of instruction words. Each individual instruction word may represent multiple individual operations, in a VLIW style. No branches are performed during the execution of a region. Instead, multiple potential branch destinations may be evaluated during the execution of the region. When the end of a region is reached the code will branch to a successor region. Thus only one actual branch is executed per region.

A region is constructed from a sequence of contiguous instructions from the original sequential representation of a program. This sequence may be subdivided into a number of "strands". Each strand is a subset of sequential instructions from the region. Every operation within a region belongs to a particular strand. Operations within a particular strand are performed in the original dependency order of the program. Operations may only be reordered if the revised order is guaranteed to produce the same results as the original. Thus loads cannot be moved across stores if there is a possibility that the locations being accessed are aliased.

The strands are numbered in terms of their original order in the sequential program representation. To maintain the original program semantics the dependencies between operations in different strands must be observed as though they were being executed in the original strand order. The strand numbering thus represents a temporal ordering for the operations with the strands. However, this strand number order does not have to be the same as the order in which operations are actually issued within the CriticalBlue code. Operations are labeled with the strand to which they belong. This is used to impose a time order of the strands that must be observed if the original code's sequential consistency is to be maintained. This strand time line is distinct from the actual execution timing of the operations.

The CriticalBlue architecture provides much greater scope for allowing the reordering of operations belonging to different strands. If necessary the dependencies between operations in different strands can be violated. The architecture contains hardware mechanisms to recover from such an event and ultimately produce the correct results. Importantly, the hardware overhead to achieve this is extremely modest in comparison to other processor architectures that support "Out-of-Order" execution.

The code generator generates additional operations that are able to detect if an inter-strand dependency has been violated. This causes two events to happen. Firstly, the higher numbered (and thus temporally later strand) is aborted so that any operations executed within it do not permanently alter the state of the machine. Secondly, when the

end of the region is reached the branch logic hardware causes the same region to be executed a second time. On this occasion all lower numbered strands are "squashed" so that the operations within them are not executed. If there are no dependency violations between strands then a region only has to be executed once. If there is a dependency violation then the region is executed a second time. Further dependency violations may require further executions. The operations (and their order) within a region are fixed and thus the processor is a simple statically scheduled machine. Out-of-order execution is achieved by re-execution of the same code while masking of the results of particular operations.

The code generator can generate a code sequence that mixes operations as appropriate from different strands. As discussed, internal strand dependencies are maintained but. If there are unused functional units in a particular instruction word then appropriate operations can be selected from other strands. In this way each of the functional units can be kept busy for a much higher percentage of the time and the instances of NOPs (and thus inefficient instruction representation) are significantly reduced. The code generator is able to directly trade off code density and performance. High code density can be achieved by violating more dependencies between strands. This requires re-executions of the region and thus overall performance is lowered. If higher performance is required then inter-strand dependencies are observed at the cost of lower code density.

The execution of each strand is conditional. An operation in one strand can cause a later strand to be squashed. If a strand is squashed then all the results it generates are nullified, as though it has not been executed at all. Many branches in the original code can be converted to use strands instead. For instance, if a branch conditionally jumps over a block of code then that code can be formed into a strand. Its execution is made conditional on the inverse of the same calculation previously used for the conditional branch.

Multiple strands within a region can be used to represent control flow of arbitrary complexity, including if-then-else constructs and conditional strands within other conditional strands. All this conditional execution can be performed using the static schedule of operations and without any complex hardware or branches that would upset the efficiency of the processor pipeline. Branches to code outside of the region remain implemented as branch operations. A number of these may be issued within a region. Branch control logic determines which branch is actually taken at the end of the region. Thus the architecture is able to evaluate a multi-way branch condition.

To improve operation parallelism a load from a later strand can be executed before a store from an earlier (i.e. lower numbered strand). Thus calculations for the two strands can proceed in parallel without serialization enforced by any potential address alias. The code generator automatically issues a special check operation. This is a simple address comparison operation between the store and the load address. If they are not the same then execution of the two strands can continue. If they are identical then the strand containing the later load is aborted. The region may then be re-executed and the earlier load is able to correctly load the value generated by the store issued later in the region. This store is temporally earlier as it is from a lower numbered strand.

4 Architectural Overview

4.1 General Philosophy

One of the key requirements of the architecture is to support scalable parallelism. The basic structure of the micro architecture and the operation of the design tools are all focused on that goal.

Extracting parallelism from highly numeric loop kernels is relatively straightforward. Such loops have regular computation and access patterns that are easy to analyse. The nature of the algorithms also tends to lend itself well to parallel computation. The architecture just needs to balance the availability of computational resources (such as adders, multipliers) and memory units to ensure the right degree of parallelism can be extracted. Such numeric kernels are common for DSPs. The loops tend to lack any complex control flow. Thus DSPs tend to be highly efficient at regular computation loops but are very poor at handling code with more complicated control flow.

Other than in numeric computation loops, C and C++ code tends to be filled with complicated control flow structures. This is simply because most control code is filled with conditional statements and short loops. Most C++ code is also filled with references to main memory via pointers. The result is a code stream from which it is extremely difficult to extract useful amounts of parallelism. In average RISC code, approximately 30% of all instructions are memory references and a branch is encountered every 5 instructions.

General purpose processors for PCs have to deal with this kind of code and extract parallelism from it in order to achieve competitive performance. The complexity of PC processors has mushroomed in recent years to try and deal with this issue. The control logic for a modern PC processor has literally millions of transistors dedicated to the task of extracting parallelism from the code being executed. The extra hardware needed to actually perform the operations in parallel is tiny in comparison with the logic required to find and control them. The main method utilised is to support dynamic out-of-order and speculative execution. This allows the processor to execute instructions in a different order from that specified by the program. It can also execute those instructions speculatively, before it knows for sure whether they should be executed at all. This allows parallelism to be extracted across branches. The difficult constraint is that the execution must always produce exactly the same answer as would result if the instructions were executed strictly one by one in the original order.

The control and complexity overheads of dynamic out-of-order execution are far too high for a CriticalBlue processor. There is a significant cost overhead due to the area occupied by the control logic, not to mention the cost of designing it. Additionally, such logic is not amenable to the scalability requirements of the CriticalBlue architecture.

A number of recent developments in the area of micro architecture have been focused on VLIW type architectures. There is a "back to basics" movement that seeks to place the burden of extracting parallelism on the compiler. The compiler is able to perform much greater analysis to seek parallelism in the application. It is also considerably simpler to develop than equivalent control logic. This is because the control logic must find the parallelism as the program is running so must itself be highly pipelined and suffers from the physical constraints of circuit design. The compiler performs all of its work up front in software with the luxury of much longer analysis time. For most classes of static parallelism, compiler analysis is very effective.

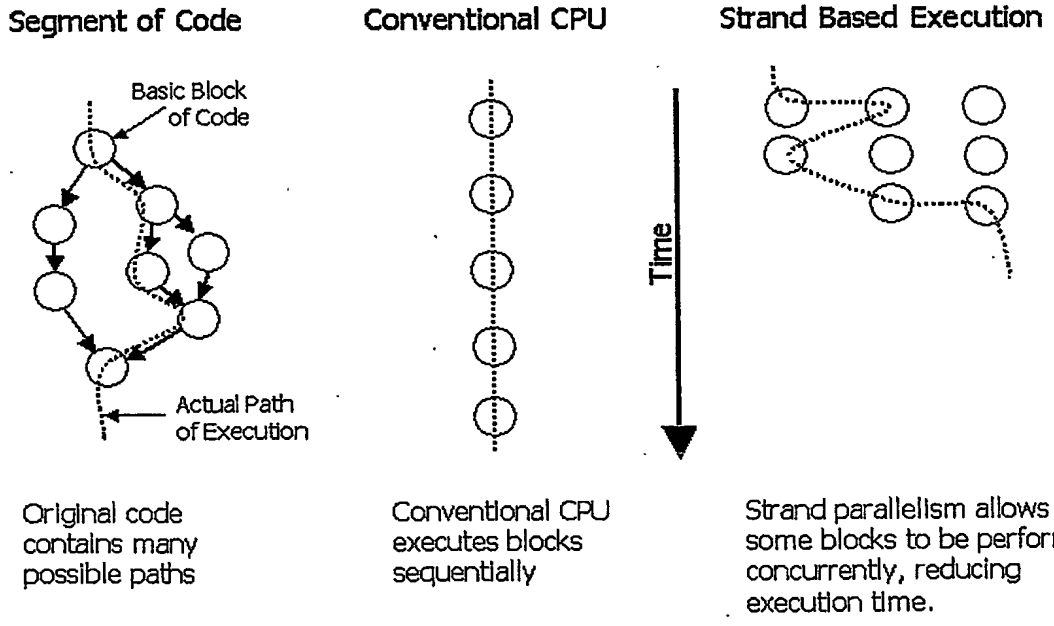
Unfortunately, software analysis is poor at extracting parallelism that can only be determined dynamically. Examples of these are branches and potentially aliased memory accesses. A compiler can know the probability that a particular branch will be taken from profiling information, but it cannot know for sure whether it will be taken on any particular instance. A compiler can also tell from profiling that two memory accesses never seem to access the same memory location, but it cannot prove that will always be the case. Consequently it is not able to move a store operation over a potentially aliased load operation as that might affect the results the program would generated. This restricts the

amount of parallelism that can be extracted statically in comparison to that available dynamically.

CriticalBlue employs a unique combination of static and dynamic parallelism extraction. This gives the architecture access to high degrees of parallelism without the overhead of complex hardware control structures. The software tools perform all the analysis, moving the complexity burden away from the hardware. The CriticalBlue architecture itself is fully static in its execution model. It executes instructions in exactly the order specified by the tools. These instructions may be out of order with respect to the original program, if the tools are able to prove that the re-ordering does not affect the program result. This reordering is called instruction scheduling and is an important optimisation pass for most architectures, and especially for CriticalBlue.

CriticalBlue has a revolutionary execution model that also allows it to perform out-of-order operations that cannot be proved as safe at code generation time. In general it only perform these optimisations if it knows that they will usually be safe at execution time. The hardware is able to detect if the assumptions are wrong and arrange a re-execution of the code that is guaranteed to produce the correct answer in all circumstances. The hardware overhead for this "hazard" detection and re-execution is very small.

The diagram below illustrates the power that this style of execution gives the CriticalBlue architecture:



An example segment of code is shown on the left hand side. This is composed of individual basic blocks. A basic block is a segment of code that is delineated by a branch operation. If execution enters a basic block then all instructions within it will be executed. At the end of the basic block there is a branch instruction that causes execution to continue with one or two different possible successor blocks. The condition upon which the branch is performed is generally calculated in the code of the basic block so it is not possible to know before entering the block which route will be taken. Each execution of the code will produce a particular path of execution through the basic blocks. Certain paths may be considerably more likely than others but any route may be taken.

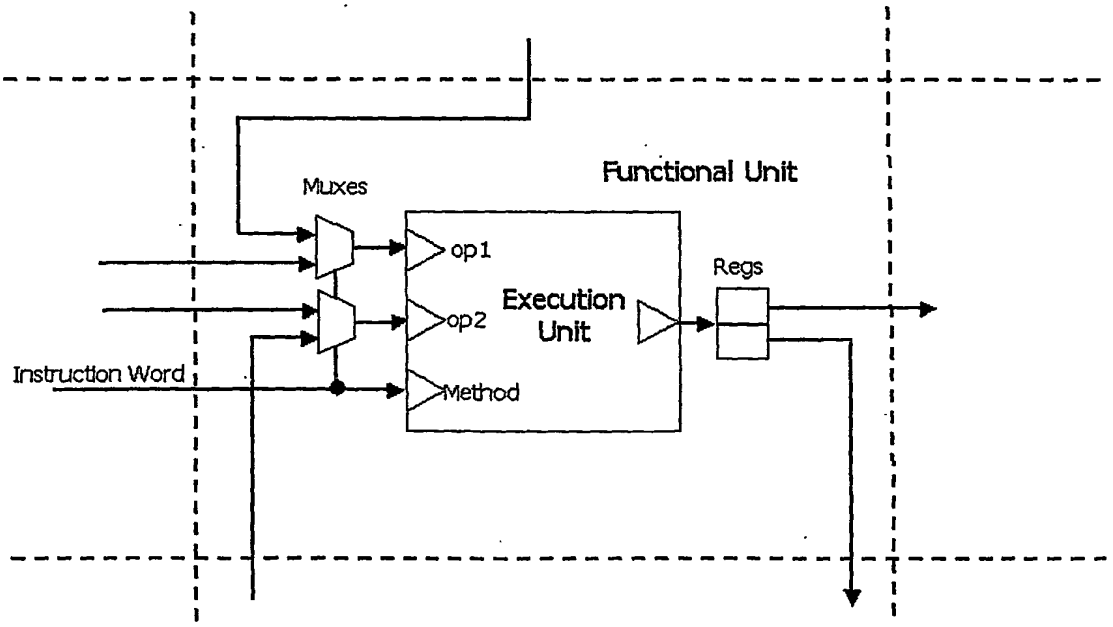
The middle section of the diagram shows the execution in a conventional processor that does not support any kind of out of order execution. This is typical of RISC processor cores. Each basic block as to be executed one after the other, as the branches are resolved.

The right hand section shows the execution model for CriticalBlue. It is able to execute code from a number of different basic blocks in parallel. It does this to increase the amount of parallelism and efficiency of the architecture. It might know that one basic block is very likely to follow another. It can pull instructions forward from the second block to execute in parallel with the instructions of the first. This allows calculations to be started earlier (and thus finish earlier) and to more effectively balance the resource utilisation of the processor. The CriticalBlue scheduling algorithm does this while taking account of the probability that the execution of a particular instruction will be useful.

Executing code speculatively in this manner normally requires a significant hardware overhead. If a particular block should not have been executed then any results it has produced must be discarded. This is referred to as "squashing" the execution. In particular, any store operations that the code has performed must be undone as they could permanently pollute the memory space with incorrect results. CriticalBlue employs mechanisms that allow the benefits of speculative execution while only requiring the minimum of hardware overhead.

4.2 Functional Units

The internal architecture of functional unit is shown below:



The central core of a functional unit is the execution unit itself. It performs the particular operation for the unit. New functional units may be created using user defined execution units. The CriticalBlue tools automatically instantiate the required "glue" blocks around the execution unit in order to form a functional unit. These glue blocks allows the functional unit to connect to other units and to allow the unit to be controlled from the instruction word.

Functional units are placed within a virtual array arrangement. The dashed lines shown on the diagram illustrate this. Individual functional units can only communicate with near

neighbours within this array. This spatial layout prevents the architectural synthesis generating excessively long interconnects between units that would significantly impact clock speed.

The control inputs include the segment of the instruction word that controls that particular functional unit. The method selector is passed directly to it. Other fields are used to control the multiplexers that select data from buses. Each operand input to the execution unit may be chosen from one of a number of potential data buses using a multiplexer. In some circumstances the operand may be fixed to a certain bus, removing the requirement for a multiplexer. The number of selectable sources and the choice of particular source buses are under the control of the CriticalBlue architectural optimisation tools.

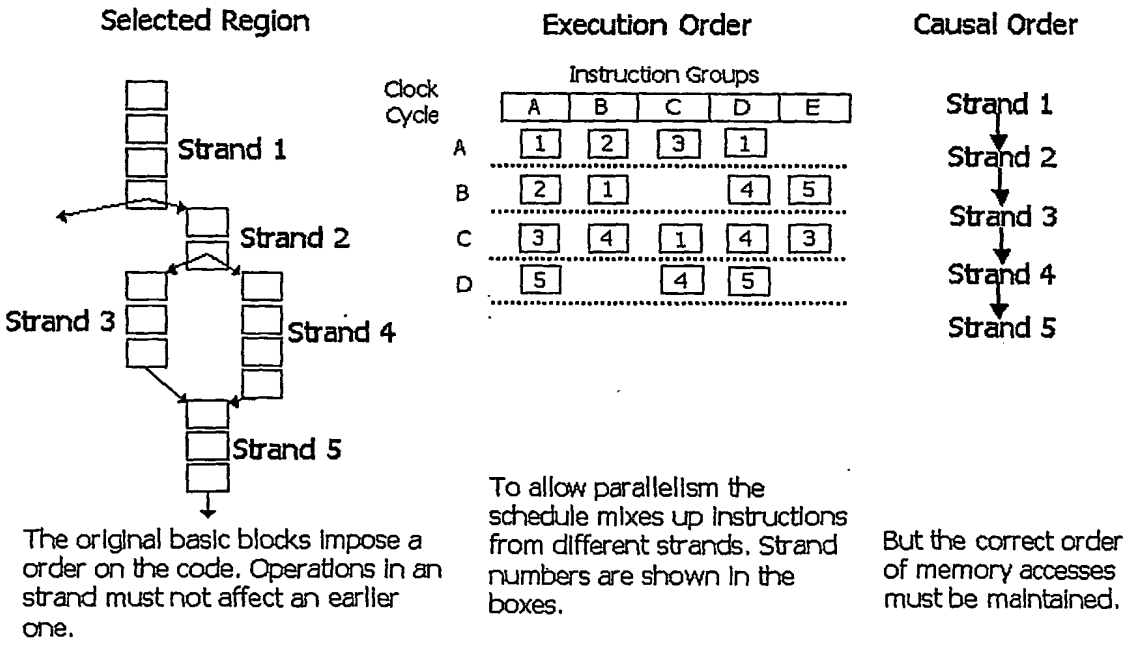
All results from an execution unit are held in independent output registers. These drive data on point-to-point buses connected to other functional units. The point-to-point nature of these buses minimises power dissipation and propagation delays. Data is passed from one functional unit to another in this manner. The output register holds the same data until a new operation is performed on the functional unit that explicitly overwrites the register.

4.3 Strand Execution Model

One of the central innovations of the CriticalBlue architecture is its "strand" based execution mechanism. These are rather like threads but represent a much lower level construct that is present in the architecture to support out-of-order execution.

A strand represents a particular sequential group of operations that is being executed on the machine. Many strands may be executed simultaneously. Each individual operation that is performed belongs to a particular strand. Whenever an instruction word is executed it may contain operations that associated with a number of different strands.

The diagram below illustrates the relationship between strands and basic blocks:



The different colours on the left hand side of the diagram represent different basic blocks in the code. The individual blocks represent the individual operations that are present in

each of the basic blocks. The last operation is a branch that determines which basic block will be executed next. Each of the basic blocks is allocated a different strand number.

The middle section of the diagram shows a potential instruction schedule generated for a CriticalBlue processor. Operations from different strands (i.e. basic blocks) may be issued during the same clock cycle. The original strand number is shown in each operation. The order of operations within a particular strand is always maintained. The order of operations between strands does not have to be maintained, allowing much greater scheduling freedom for the architecture. Although the scheduler is able to perform operations out of order between strands it will only do so if that is unlikely to lead to a hazard. The hardware is able to recover from a hazard but there is a performance penalty to doing so.

This mechanism allows instructions to be issued out of order. However, if the correct results are to be produced by the architecture then the data flows between strands that would occur if they were executed in the correct order must be maintained. The right hand part of the diagram shows that the logical order of the strands is always the same. A result generated by an operation in strand 3 should never influence the calculations performed in strand 2. That could never happen if the instructions were executed in order. In effect the architecture has two different time domains. There is the physical time domain of the order of instruction execution. There is also the logical time domain that was present in the original program that must be preserved in order to maintain the original program semantics.

The CriticalBlue tools can determine the correct ordering of most operations statically. The main exception to this is memory operations, where the addresses cannot be determined at compile time.

4.4 Region Based Execution

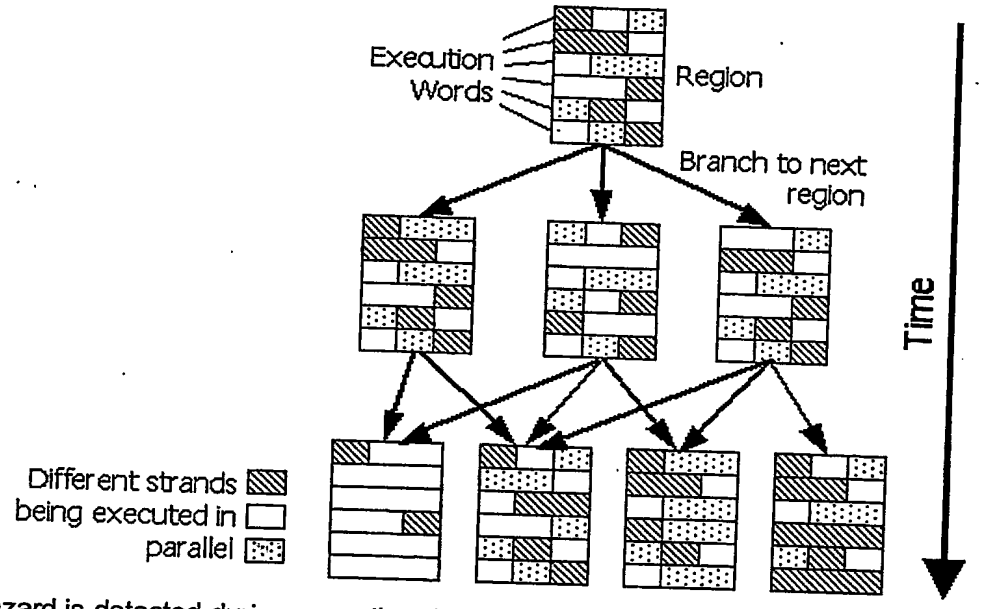
In the CriticalBlue architecture all execution is performed within blocks of code called regions. This simplifies the implementation of both the instruction scheduling and the strand control mechanisms in the hardware.

A region is a block of code that only has a single entry point but potentially many exit points. The analysis performed by the CriticalBlue tools is able to form groups of basic blocks into regions. Regions are often used as the basic arena in which global scheduling optimisations are performed. Global scheduling refers to the movement of instructions across branches as well as within individual basic blocks. Global scheduling is a considerably more difficult problem than basic block scheduling.

In the CriticalBlue architecture, regions are always executed fully. If the region contains a number of internal branches to basic blocks outside of the region then they are not resolved until the end of the region reached. The compiler constructs the regions from basic blocks so that they contain the most likely execution paths through the basic blocks. A region is able to perform a multi-way branch to select one of a number of different successor regions.

All strands are limited to the lifetime of a single region. The architecture is able to execute operations out of order within a particular region. Out of order execution and any resulting hazards are resolved at the end of the region and then execution continues on to another region, which may itself issue operations out of order.

The diagram below illustrates an example set of regions and the relationships between them. It shows the execution of the individual strands within each region:



If a hazard is detected during execution then the sequential semantics of the strands have not been properly preserved. The architecture must be able to recover from this situation with as little overhead as possible.

Upon detecting a hazard in a particular strand the results generated for that and any later (i.e. higher numbered) strands may be incorrect. The architecture allows execution to continue until the end of the region, when the strands will be completed. Any results from the hazard, and any higher, strands are discarded. The architecture then re-executes the code from the start of the region again. Since lower numbered strands have already been successfully completed they are not executed a second time. The architecture includes logic to block operations from those strands. Since the lower strands have completed and generated their results the hazard strand is able to execute correctly, utilizing any required results from the lower strands. If another, even higher numbered, strand generates a hazard then the region may be repeated a second time. When all strands have successfully completed the processor may move onto the successor region.

Of course, the goal of the CriticalBlue architecture is to execute all strands successfully on the first attempt. The compiler does extensive analysis to ensure that the chances of hazards are small. The key is that the compiler doesn't have to prove that a hazard cannot happen. The re-execution mechanism will ensure correct completion of the strands if required. It does this with a minimum of hardware overhead. The size of regions is limited to a few tens of instructions so that the overhead of any re-execution of the region is not too great.

5 Execution Model

5.1 Key Requirements

One of the key requirements of the architecture is to support a highly scalable execution model. The architecture is a flexible as possible in terms of how individual execution units may be implemented. CriticalBlue effectively "abstracts away" everything but the fundamental control structures that need to be present to allow the execution of code.

A traditional embedded RISC processor has a rather fixed structure, even if the user may configure some aspects. The instruction format, the registers and the pipeline for instruction execution are all fixed. CriticalBlue does not fix any of these attributes. They are completely under the control of the user via the design tools.

There are two key aspects to scalability:

- **Physical Scalability:** This refers to the ability of the architecture itself to scale without being restricted by control or data bottlenecks. A scalable architecture allows more execution units to be added to fully utilise the available parallelism in an application. This must be achieved without the control of the execution units and their connectivity constraining the overall system performance. In a manually designed processor there will generally be a number of significant bottlenecks in the design. The designers will carefully balance these so that no single bottleneck has an overly detrimental impact on the architecture. Unscalable control or data flow structures tend to grow exponentially in size, or detrimental effect on overall clock speed, as additional execution units are added to the system. The CriticalBlue architecture avoids such structures.

- **Logical Scalability:** This refers to the ability of the architecture to extract available parallelism from the application being executed. There is no value in an architecture that has good physical scalability if it is not possible to produce a tool flow that is able to make good use of that capability. The CriticalBlue tools use advanced analysis to extract as much parallelism from the input machine code. This allows the user to specify their algorithms in an easy to code and understand sequential manner. The tools extract the parallelism from the application automatically to provide an efficient and scalable mapping to the architecture. The problems associated with handling potential memory aliases are one of the biggest constraints on parallelism in previous systems. CriticalBlue incorporates some innovative techniques to circumvent these restrictions.

The underlying micro architecture of CriticalBlue provides excellent physical scalability. The compilation and code generator tools provide equally good logical scalability to extract high degrees of parallelism from an application. Uniquely, the CriticalBlue design tools fully integrate the analysis of the physical and logical scalability. That is, the tools synthesis an architecture that is application specific and appropriate to the parallelism available from the application. This ensures that the physical and logical scale of the system always remain in good balance.

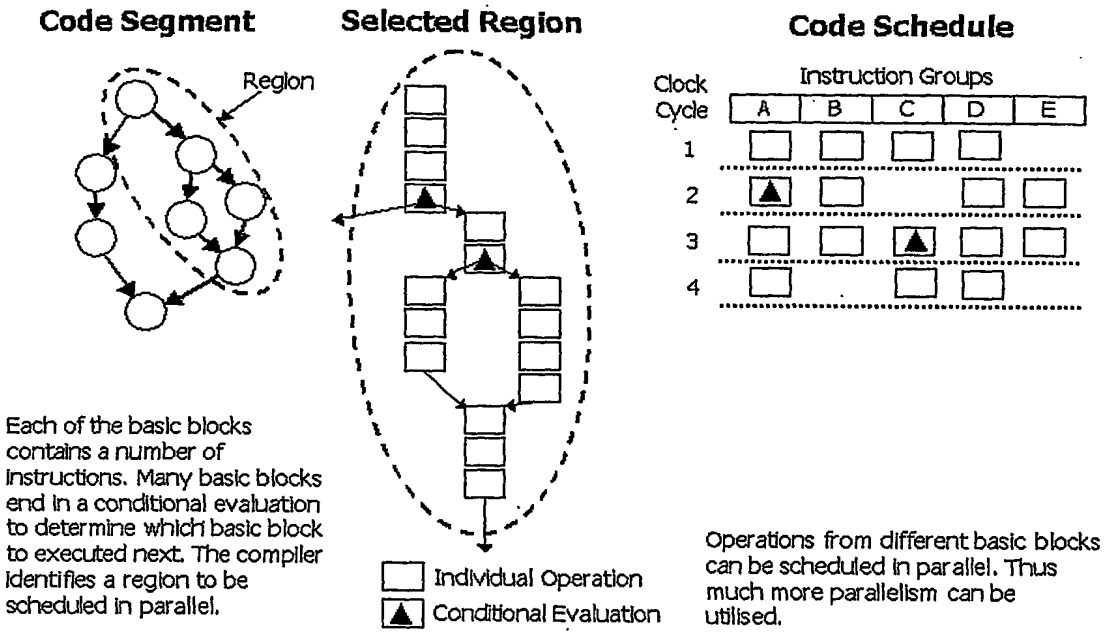
5.2 Division into Regions

All code generated for a CriticalBlue processor is within a region. A region is a group of consecutive execution words to be executed. Execution of a region may only enter with the first execution word and may only exit at the last execution word (save for temporary interruption by an interrupt handler). Thus all words within a region are always executed if the region is entered. This property considerably simplifies the code generation and scheduling process.

Within a particular region a number of branches may be issued. The branches do not have any effect until the end of the region is reached. At that point the hardware is able to resolve between many possible following regions. Thus the hardware handles a multi-way branch. Collapsing multiple branches into a single decision point allows the architecture to execute code efficiently without the need for complex branch predication mechanisms.

5.3 Division into Strands

Within each region, operations are further subdivided into a number of strands. Each operation belongs to a particular strand. The diagram below illustrates the construction of strands within a region:



The diagram shows a code segment being selected to form a region. This region may itself consist of multiple basic blocks and therefore multiple branches. Thus some of the code within the region is conditional since it would not be executed if certain branches take particular courses. The selected region is shown in more detail in the centre of the diagram. The individual operations from the basic blocks are shown with the condition evaluations for the branches shown with an inserted triangle. In this diagram each basic block is allocated a different strand. All the operations within a basic block belong to the same strand. The CriticalBlue schedule on the right hand side of the diagram shows the individual operations scheduled for execution. Multiple operations may be issued on each clock cycle and these may come from a number of different strands. The scheduler is able to perform "global scheduling" where operations are moved between basic blocks in order to lower execution time and make good use of functional unit resource availability. The conditional evaluation operations can be scheduled like other operations. This allows a great deal of flexibility in the arrangement of code.

The strand mechanism performs two main purposes:

- Firstly, it implements a predicate mechanism. This allows conditional blocks of code to be integrated into a region in which all operations are executed. Each strand has an associated predicate flag. By default this is set to true but the strand may be "squashed" and this flag set to false. If a strand is squashed then all operations from the strand are essentially nullified as if they were never executed. Thus it provides a means to incorporate conditional code into an atomically executed region. Thus the pipeline stall hazards of conditional branches are avoided.

- Secondly, it provides a framework to implement a low cost speculative execution mechanism. The code scheduling may make code motions that violate the

operation ordering rules for inter-strand dependencies. The hardware incorporates simple detection mechanisms to determine when a dependency hazard has actually occurred. The numbering of the strands is in accordance to the temporal ordering of the operations in the original program. This temporal ordering may be at odds with the physical ordering in a code scheduling if some types of speculative code motions have been performed. Since each operation is with the strand to which it belongs it is possible to determine the correct temporal order of operations without any additional hardware analysis or complex instruction reorder buffers. Recovery from a dependency hazard is relatively easily achieved by re-execution of a region. This is done a number of times with certain strands squashed. This allows the original temporal order to be reproduced.

The order of operations within a given strand is fixed by the dependencies between operations within that strand. If two operations may depend on each other (and thus their order can effect program results) then the same order must be maintained in the final schedule. The dependency rules between strands can be more flexible, however. In some circumstances operations that are potentially dependent can have their order transposed.

There is a loose correspondence between strands and basic blocks in the original program. However a single basic block can be transformed into multiple strands in some circumstances. This happens if a basic block contains certain types of instructions or is beyond a certain length. Moreover, conditional branches that are normally represented as a single instruction are broken down into two operations in the CriticalBlue architecture. These are the condition evaluation and the branch itself. The branch is considered to be located in a separate basic block from the condition evaluation.

5.4 Strand Phases

5.4.1 Philosophy

Each strand has two distinct phases of execution that occur during the execution of a region.

Each strand initially enters a speculative phase. During this phase operations from the strand may be executed that are speculative. An operation can be both control and data speculative. An operation is control speculative if it is executed before it is known whether the strand to which it belongs should be executed at all. An operation is data speculative if it is operating upon data that might not have its correct value as expected at the start point of the strand in the program. Operations issued in the speculative phase of the strand may have to be re-executed any number of times. Thus they may not permanently change the state of the machine.

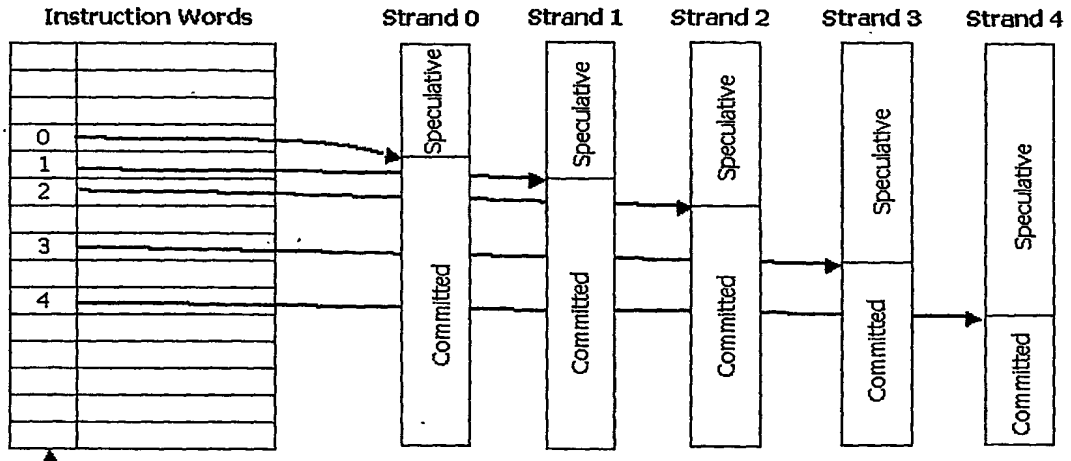
At some later point each strand enters a committed phase. During this phase it is known that the strand is to be executed and the entry conditions in terms of register and memory state are correct for the strand's position in the code. Operations that permanently change the machine state may be executed during the committed phase.

The division of strands between speculative and committed phases allows speculative operations to be performed that can be squashed without permanently changing the machine state. This gives much greater scheduling freedom and thus potential parallelism. Moreover, the mechanism requires very little hardware overhead.

5.4.2 Changing Strand Phase

A strand changes state from its speculative to committed phase if a commit operation is issued for it. A special commit unit executes a commit operation. The number of the strand to be committed is specified as part of the operation.

An illustration of the transition from speculative to committed phase is shown in the following diagram. The strand numbers for the commit operations are shown in the instruction word.



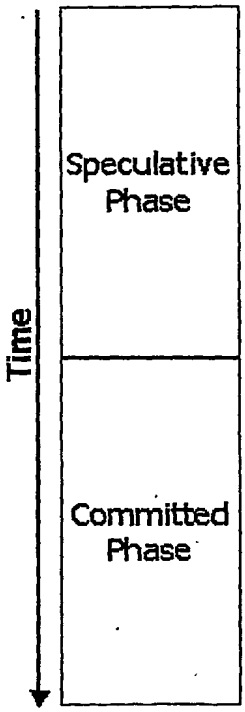
Strand numbers specified to commit units. The strand enters its commit phase on the next cycle.

Generally the transitions from speculative to committed state are in strand order (i.e. a strand can only be committed if all lower numbered strands are also committed). Out of order transitions are possible if the strands are guarded such that they only commit if all previous strands have been previously committed.

A strand immediately enters its committed phase once a commit operation has been issued. Thus operations issued on the following clock cycle are in the committed phase.

5.4.3 Phase Requirements

There are certain restrictions on the types of operations that may be issued in particular phases of a strand's execution. Certain criteria must also be met before a commit operation may be issued to transition the strand into its committed phase. This is illustrated on the diagram below:



Speculative Phase Requirements

- ☐ Predication flag in SPM does not have to be resolved
- ☐ Branches may be executed (to be resolved later)
- ☐ Guard or Check Hazard instructions may be executed
- ☐ Memory writes may not be issued (except when using a SWB)
- ☐ Register writes may not be executed
- ☐ Volatile and squash instructions may not be executed

Phase Transition Requirements

- ☐ Predication flag in SPM for strand must be resolved
- ☐ All branches for the strand must have been executed
- ☐ All previous strands must have committed

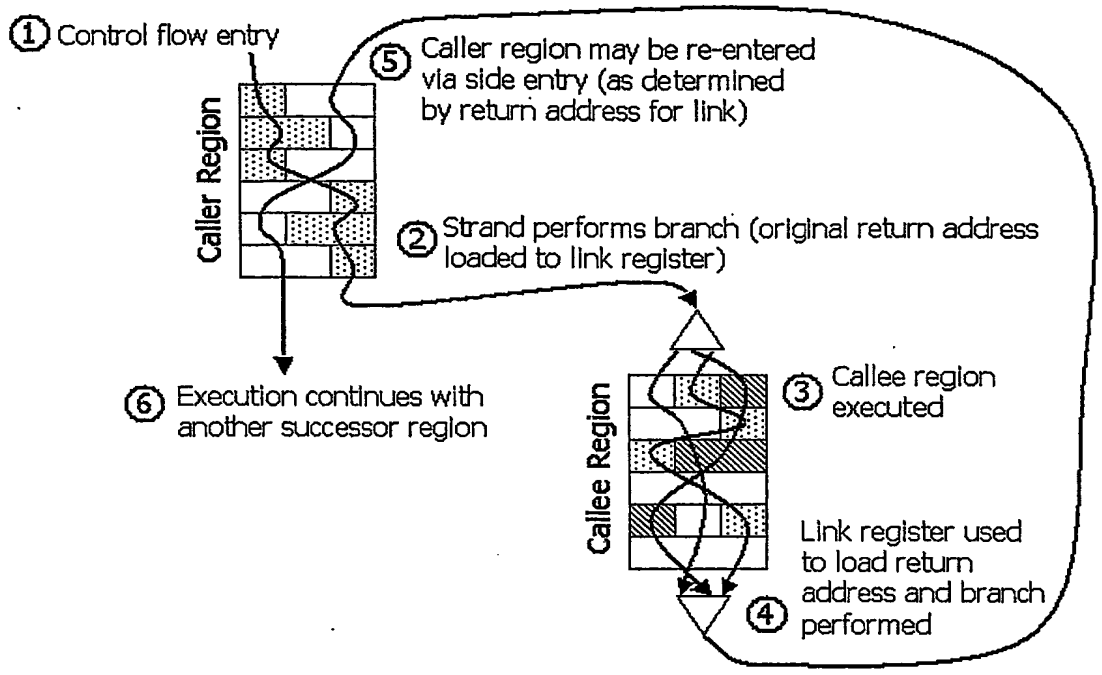
Committed Phase Requirements

- ☐ Memory writes may be executed
- ☐ Register writes may be executed
- ☐ Volatile and squash instructions may be executed
- ☐ Guard or Check Hazard instructions may not be executed

5.5 Subroutines

5.5.1 Calling Mechanism

The subroutine calling mechanism is illustrated in the diagram below. The mechanism does not require any hardware support and allows full compatibility with the host architecture. The return address that is held is actually the return address for the host architecture and no additional stack frame storage is required. Thus the mechanism is fully compatible with host debuggers that read return addresses on the stack frame in order to generate a stack trace back.



A call is implemented as a standard branch on the architecture. Before executing the branch the host link register is loaded with the return address. This is the address of the instruction following the call in the original code. This is normally performed by the program counter being copied to the link register as a side effect of a call. In the translated CriticalBlue implementation the link register is loaded with an immediate value that exactly corresponds to the value that would be loaded in the host implementation.

The called region is executed. The link register may be preserved on the stack frame if further calls are to be made. When the end of the called function is reached (which may or may not be in the region initially called) a return instruction from the host architecture is encountered. This is effectively an indirect branch using the link register as a destination. This indirect branch is converted into an indirect load and then branch by the code translation. Thus the data at the link register address is loaded. This is used as the destination for a branch. The location loaded is the original address of the instruction following the call. The code translation process places a native code address in that location.

The location contains both a region address and a first strand number. The generally occurs to the same region that originally initiated the call. However, the strand number is set to one more than the strand that caused the call. Thus the calling strand and any earlier strands are squashed. In some cases where performance is highly critical the return may be made to a different strand that does not contain earlier squashed strands.

5.5.2 Parameter Passing

Parameters are passed to functions using the standard calling conventions for the host architecture. The first few parameters may be loaded into fixed registers and higher numbered parameters loaded to particular areas of the callee stack frame.

5.5.3 Parameter Return

Parameter returns from functions use the standard value return conventions of the host architecture. Normally a return result is placed in a specific register.

6 Read Speculation

6.1 Observing Memory Dependencies

One of the most difficult aspects of the architecture is maintaining memory dependencies. These must be preserved between memory accesses that might potentially point to the same memory location (i.e. they may be aliased). If the compiler is able to prove that two memory accesses cannot be to the same location then they may be arbitrarily reordered without affecting the results generated from the program.

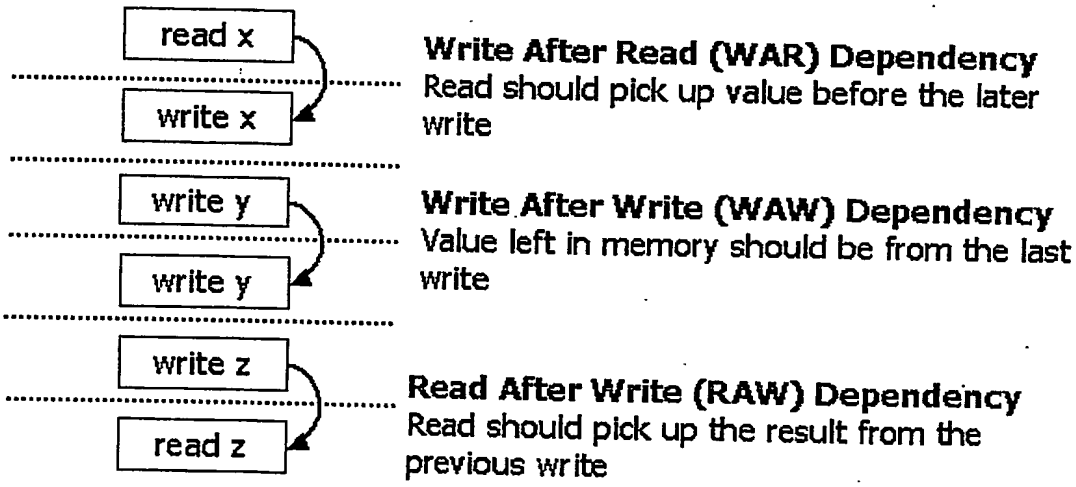
The code generator must observe dependencies between memory accesses that may be to the same location. Such accesses are said to be aliased.

The dependencies that must be observed are as follows:

- **Write-After-Read (WAR):** Writes must not be scheduled earlier than a preceding read that may be aliased
- **Write-After-Write (WAW):** The order of potentially aliased writes must be maintained.
- **Read-After-Write (RAW):** Reads must not be scheduled earlier than a preceding write that may be aliased

Write operations may only occur in the committed phase of a strand. Maintaining these rules significantly restricts the amount of parallelism available to the architecture.

The following illustrates the different classifications of memory dependencies:



6.2 Improving Scheduling Freedom

Software analysis is unable to derive much information for a large proportion of memory accesses. These would have a major detrimental effect on the potential parallelism in the architecture. Fortunately the CriticalBlue architecture uses the strand mechanism to allow these to be executed out-of-order in many circumstances.

Dependencies between memory operations within a strand are always preserved. That is, the order of such instructions is not changed. The scheduler also maintains WAR and WAW dependencies for memory operations between strands.

The majority of performance benefit can be obtained by allowing RAW dependencies to be reordered. This allows a read to be scheduled earlier than a potentially aliased write. Most chains of calculations start with a read operation, and this allows the calculations to be started earlier and be potentially executed in parallel with other code. If the read and write are not aliased then the correct result is produced. If, however, they are aliased then the read should have obtained the data stored by the write. The whole chain of calculations will produce an incorrect answer.

Whenever a potentially dependent read is moved before a write the scheduler also generates a checking operation. This compares the addresses of the read and the write. If they are identical (i.e. there is a hazard) then the strand containing the read is aborted. The earlier strand containing the write is allowed to complete and then the whole region is repeated. The read is then able to pick up the value written by the earlier strand and the correct result is produced. This whole process requires the minimum of hardware overhead. No additional compensation code is required to recover from a hazard situation. The underlying microarchitecture and execution model handle it automatically.

6.3 Check Hazard Instructions

Check hazard (chaz) operations are used to validate the boosting of loads over stores. They must be inserted whenever a read has been boosted over a previous write. It checks for an address conflict and aborts the reading strand if so. All chaz instructions must be issued in the speculative phase of the reading strand while all writes must be issued in the writing strand's committed phase.

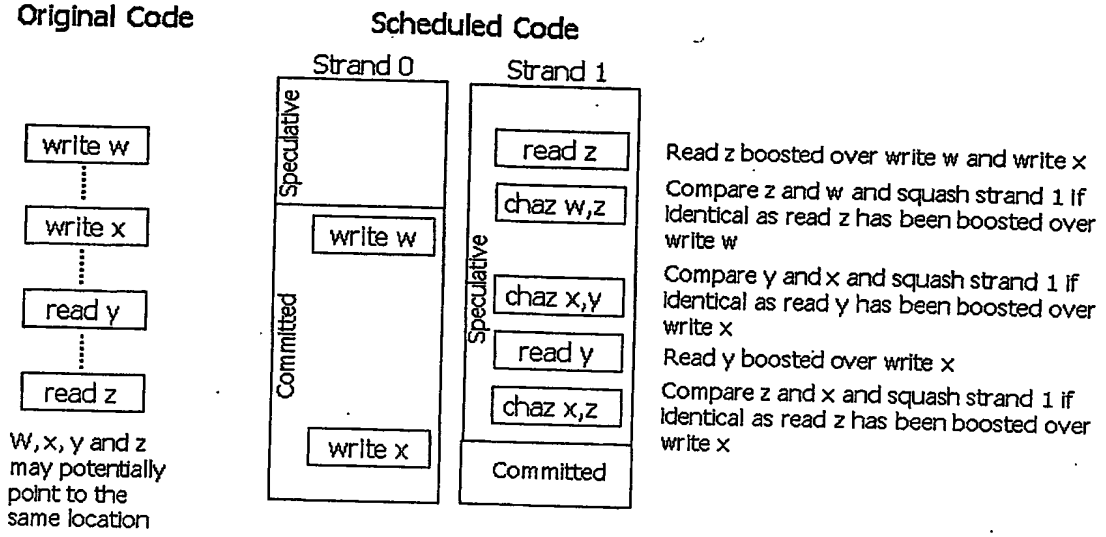
A particular processor may support a number of individual chaz units. A chaz unit simply compares two addresses and generates an abort if they match. This is used to compare the load and store addresses and abort the load strand if there is an address match.

A chaz operation has two operands, a left operand that is the store address and the right operand that is the load address. The execution strand number is obtained from the left operand. If the store strand is not being executed then the operation is disabled.

The number of the strand to be aborted is obtained from the strand associated with the right hand operand (the load address).

6.4 Read Speculation Example

The diagram below shows an example of read speculation being performed.

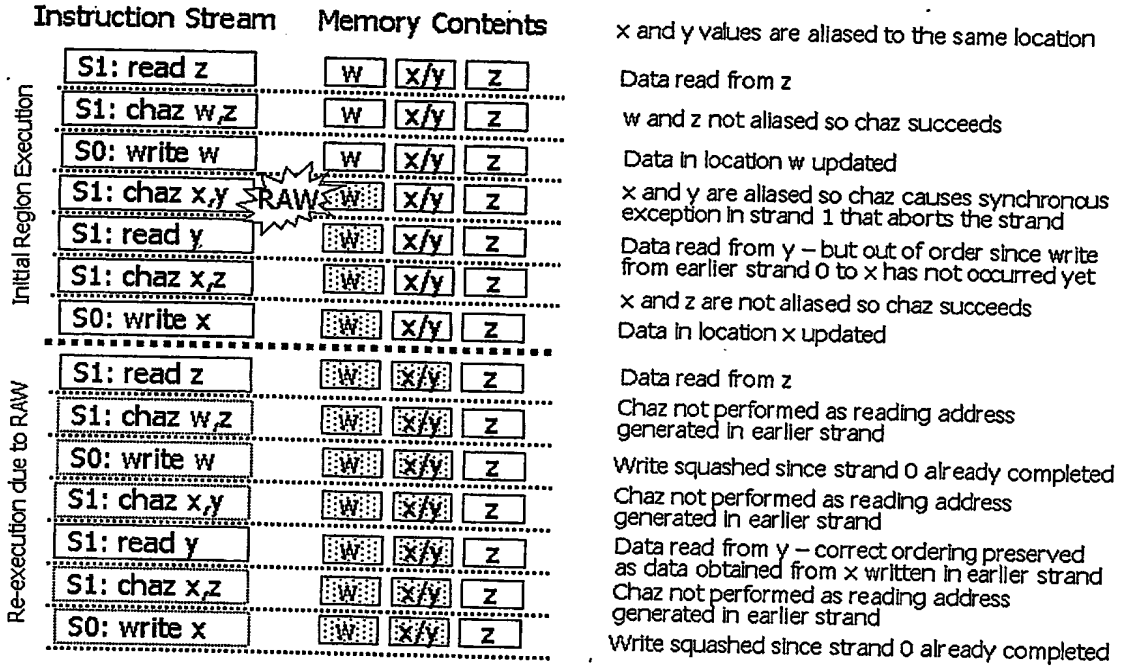


The original code is shown on the left hand side. Two write operations are followed by two read operations. Any of the four addresses may in fact be aliases to the same memory address. Software analysis before execution is not able to determine if those locations could be aliased or not.

The scheduled code is shown on the right hand side. The operations are split into two distinct strands. If read speculation is being performed then it must be done on an inter-strand basis. Speculation between accesses within a strand is not possible as there is no way for the hardware to perform the required selection re-execution of code. The two write operations are placed in the temporally first strand 0. This is because the writes are before the reads in the original code. The reads are placed in strand 1. However, the placement into different strands gives the code scheduler the freedom to move the reads earlier than potentially aliased writes. The read z is moved earlier than both writes, and the read y is moved earlier than the write x. This boosting allows improved scheduling freedom and thus greater parallelism and performance.

If any of the locations are indeed aliased then this code sequence will produce incorrect results. Thus chaz operations must be inserted to detect hazards and initiate the recovery from them. A chaz is required for each write that a read is moved across to which it may be potentially aliased. The read z has two chazs for comparing against both the w and x values. The read y has a single chaz for comparing against the x value.

The diagram below shows an example execution of the same code. The instruction stream shows the operations being performed and the memory content shows the value of memory locations at that point of execution. The operations are tagged with the strand that they belong to. The memory contents is coloured differently as the value in memory is updated by a write operation.



In this case the memory locations x and y are aliased (i.e. actually point to the same memory location). Thus when the chaz instructions comparing the x and y values is executed a hazard is detected, as shown on the diagram. Since the chaz is in strand 1 all operations in strands 1 or higher are disabled from that point of execution. Since all chazs for a strand must occur before any writes (since chazs are in the speculative phase and writes can only be performed in the committed phase) the strand will not have changed the memory contents. Despite the hazard execution continues until the end of the region. The values of locations w and x/y are updated by the writes in strand 0.

When the end of the region is reached the previous hazard causes the region to be re-executed. If no hazard was detected because there was no aliasing then all strands would have been successfully completed during the initial execution. On the re-execution, due to the RAW hazard, strand 0 is disabled since it was previously completed. Thus the writes from strand 0 are not illegally repeated a second time. The x and y locations will still be aliased on the second execution. However, the chaz comparison will not be performed since the x value is computed in strand 0. Since that strand is not executed the chaz will detect the strand tag from the x value and disable the chaz. Thus the chaz will not cause a hazard on the second execution. The read y operation is performed and reads the value written by the write x operation on the first execution of the region. Thus even though the write x operation occurs after the read y operation in the region schedule, the re-execution allows the later write to pass data to the earlier read if necessary.

7 Strand Control Units

7.1 Strand Control Unit

This controls the execution of strands within a particular region. The unit generates a vector of the current strands that are being executed. Functional units use the vector in

order to squash operations from disabled strands. The unit also detects conditions that require the current region to be re-executed to maintain memory dependencies.

7.1.1 Current Instruction Count (CIC)

The CIC holds a count of the current instruction word being executed from the region. This is used to determine the position for restarting execution upon return from an interrupt. An interrupt causes an immediate branch to the interrupt handler before the region has completed execution.

When an interrupt handler is being entered the CIC is copied into the Restart Instruction Counter (RIC). During the same cycle the values in output registers are copied into their corresponding shadow register. Other registers that are preserved during an interrupt such as RBA and SPM are also stored in shadow copies.

A count is used rather than an absolute address in the instruction buffer since the interrupt handler may displace the region from the buffer. Thus it may be loaded to a different address when execution is resumed.

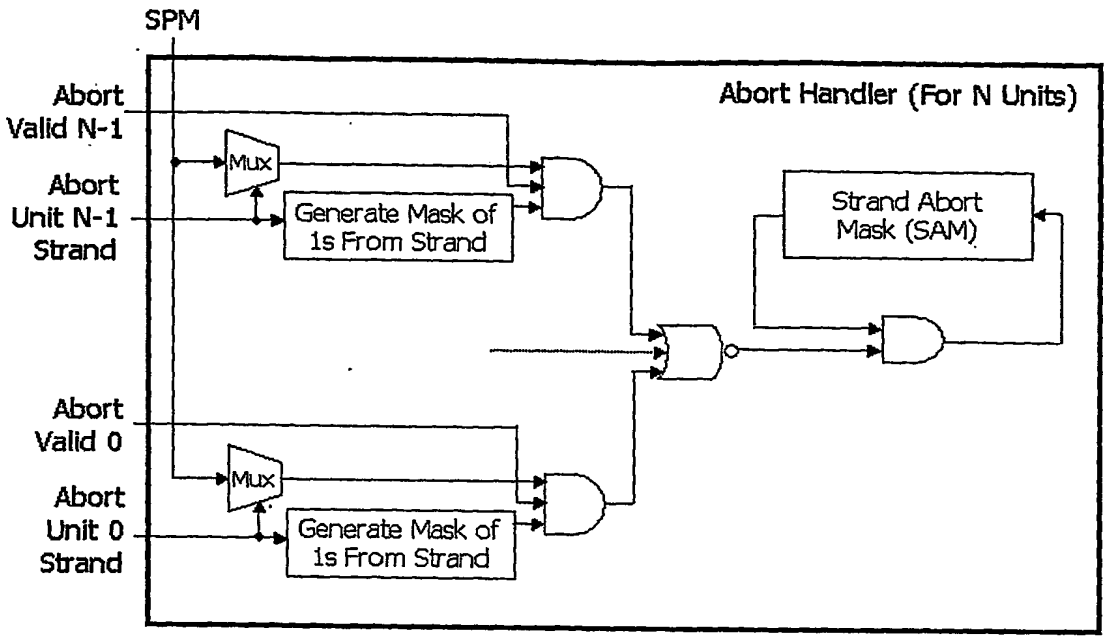
7.1.2 Restart Instruction Count (RIC)

The RIC holds the count of where instruction execution should be restarted following a return from an interrupt. When a return is made the interrupted region is re-executed from its first instruction again. This is necessary since the region may have to be reloaded into the instruction buffer if interrupt handler has displaced it. Initially the SEM for execution is forced to all clear so that no instructions The RIC is compared against the CIC register and when they are equal the SEM is restored to its normal value. Thus execution of operations continues normally from that point. At the same time the shadow values stored in output registers are restored into the main output register.

7.1.3 Strand Abort Mask (SAM)

The SAM is a bit mask with a single bit for each strand. If a bit is reset then that indicates that the corresponding strand has been aborted. A strand may be aborted due to a check hazard or guard instruction. A strand abort indicates that the there has been a dependency violation and the strand must be re-executed (unless the strand is squashed).

A number of individual abort buses are supported to allow the use of multiple units that may generate strand aborts. The logic for updating the SAM in response to these aborts is illustrated below.



Each abort unit produces a strand number to be aborted. The appropriate bit from the SPM is examined. If the strand is already squashed then no action is taken. If the strand is not squashed then that strand and all higher numbered strands are aborted. Higher numbered strands have to be aborted since, if a strand produces an incorrect value due to a dependency violation, then higher strands may be polluted with the incorrect value. The valid flag from the abort unit gates the strand mask. All the aborts are combined and are used to make off bits in the SAM.

Note that is a strand is aborted but later squashed then higher numbered strands will have already been aborted. This may cause a region restart that is unnecessary. Thus it is generally better to resolve the squash status for a strand before issuing any aborts for it.

Upon entry to a region the SAM is initially cleared to all 1s to indicate that no strands are aborted. At the end of region execution the SAM is combined with the SPM to determine if any un-squashed strands were aborted. If so then the region is restarted.

7.1.4 Strand Predicate Mask (SPM)

The SPM is a bit mask with a single bit for each strand. If a bit is reset then that indicates that the corresponding strand has been squashed and it should not be committed. Bits within the SPM are cleared by squash operations. A single squash operation may clear a block of bits within the SPM. Once a bit is reset it may not be set once again for the duration of the region execution.

A squash is able to clear multiple bits in the SPM in order to efficiently support nested conditional constructs. The top level strand must squash all the strands within the conditional construct if the top level conditional is false. This is because squash operations issued in squashed strands are disabled. Squash operations themselves may only be issued in the committed phase of their home strand, as they cannot be executed speculatively since they have an effect on SPM.

The branch control unit may also generate squashes. Such squashes are the result of branch resolution. If a branch from a particular strand is to be taken then all higher numbered strands are squashed. This prevents their execution as they are not logically reached since an earlier branch is taken.

The bit with the SPM for a particular strand should be reset before that strand enters its committed phase. It is illegal to squash a strand when it has already entered its committed phase. The code generation ensures that this rule is not broken.

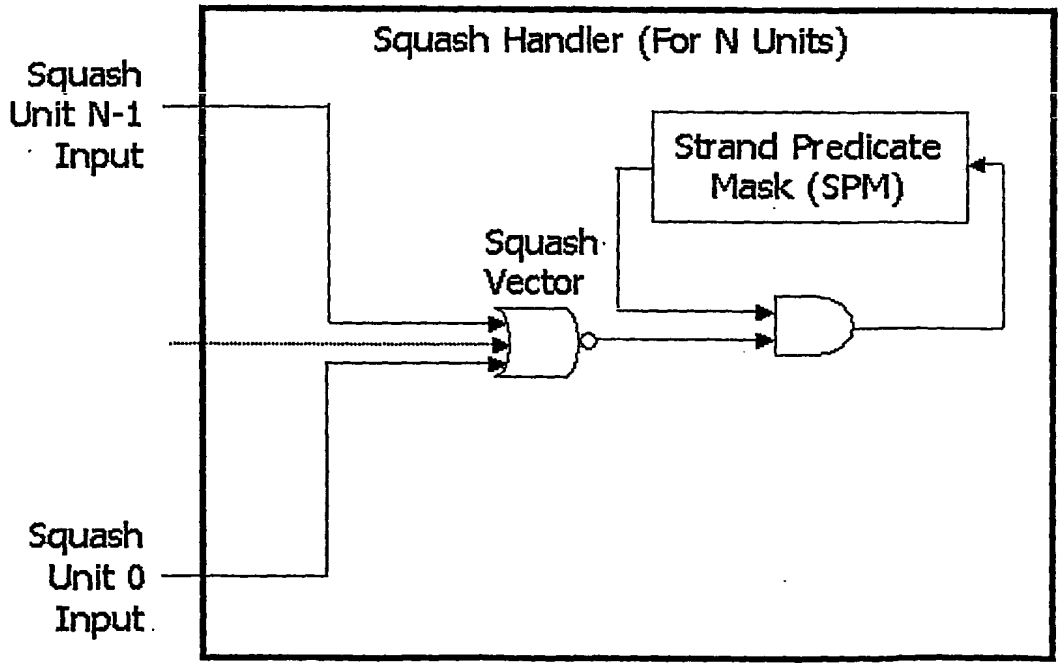
When a region is first entered the SPM is set to an initial state depending upon the entry strand. All strands below the entry strand has their SPM bit cleared to squash them. All higher bits are set so that the strand is enabled. Normally the entry strand number is 0 so that all bits in the SPM are set. A non-zero entry strand may is used when entering a region through a side entry. Side entries are used to support returns from calls within a region. This mechanism allows the return to be made to the start of the calling region. The calling and all earlier strands are disabled so that they are not re-executed.

When the end of a region is reached the SPM is logically ANDed with the logical NOT of SAM. Thus if no strands where aborted then the SPM is cleared. However, any strands that were aborted maintain an unchanged SPM state. Thus if there are any un-squashed but aborted strands then they will have the corresponding bit set in the SPM. If the SPM is non-zero then the region is restarted in order to execute those strands.

A shadow SPM register is maintained. This is used to hold the SPM state if an interrupt handler is entered. The main SPM contents is copied to the shadow on entry to the handler and restored upon exit.

The SPM is updated on each cycle on the basis of squash vectors generated by the squash units in the processor. A number of separate squash units may be supported in order to provide greater parallelism in the processor. Each squash unit provides a vector of strands to squash. These are combined and are then used to clear bits in the SPM. The SPM is read on each cycle in order to form the SEM. The SEM shows which strands are currently enabled and is distributed to all functional units. It used to disable operations issued for disabled strands.

The logic for updating the SPM is shown below:



7.1.5 Strand Enable Mask (SEM) Calculation

The Strand Enable Mask (SEM) is calculated on each clock cycle and distributed to all functional units. It is used to mask all operations issued to strands that are disabled. This prevents disabled strands generating permanent side effects from their committed phase.

The SEM is calculated from a number of individual components:

- **SPM:** Any strand that has been squashed is disabled. Once a strand has been squashed then no further operations are executed from it. There is a delay of a few clock cycles between issuing a squash operation and the effect being represented in the SEM value distributed to all functional units. The squash must therefore be calculated some clock cycles ahead of the affected strand entering its committed phase.

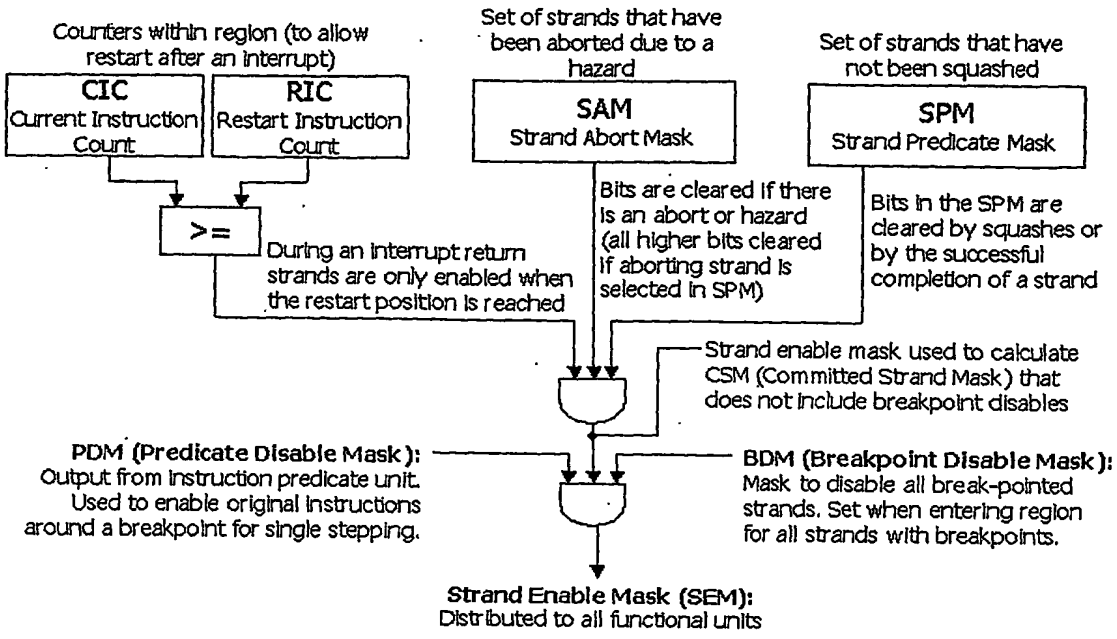
- **SAM:** Any strand that is marked as being aborted is disabled. The SAM is temporary and is reset if the region is re-executed.

- **Restart Detection:** This is used to handle the special case of a return from an interrupt. An interrupt halts the execution of a region at an arbitrary point. The RIC counter shows the instruction where execution should resume. All strands are disabled until the restart instruction is reached.

- **Instruction Predicate:** An output from the instruction predicate unit is used to control the execution of all strands. This allows individual original instructions to be executed in shadow code.

- **Break-Pointed Strands:** This is used to disable all strands for which breakpoints are set within the region. The strand control unit contains a number of breakpoint registers. Each of these has a strand base address and the strand within the region in which the breakpoint is located. If that strand is committed (i.e. it is not squashed) then a breakpoint has been reached. The synchronization branch is initiated to advance execution to a particular original instruction within the shadow code. No operations from the strand within the main code region should be executed.

The calculation of the SEM is detailed in the diagram below:



7.2 Branch Control Unit

The branch control unit determines which region will be executed next. The unit is able to handle multi-way branch conditions. A number of branch destinations with associated conditions may be issued in a region. The branch control unit determines which branch will be taken on the basis of which conditions evaluate to true and the relative priority of the branches. The unit also controls the initiation of handler functions for interrupts.

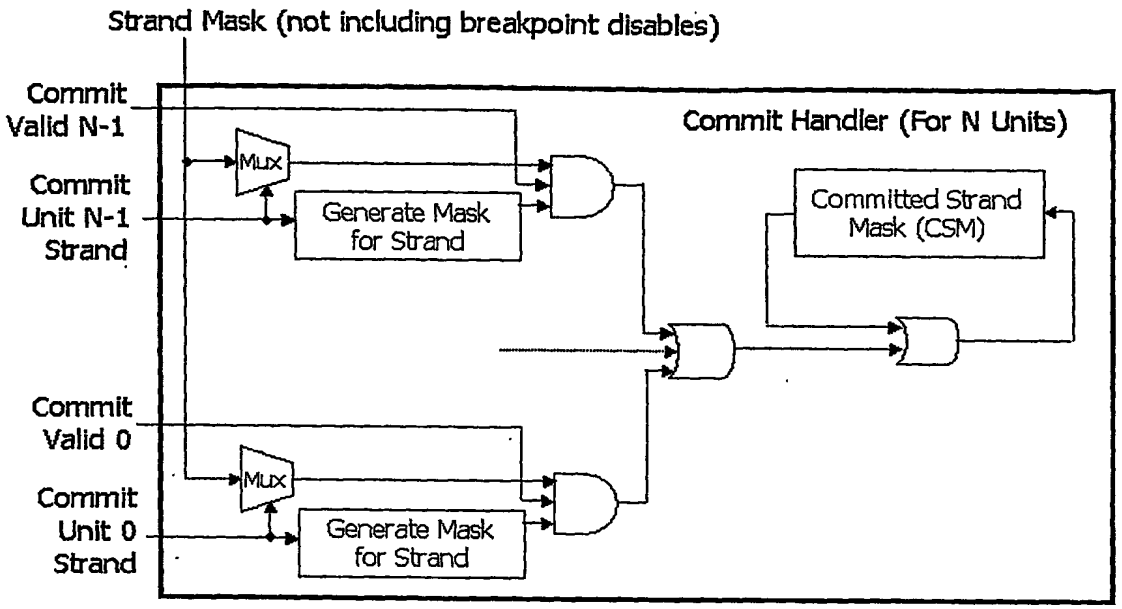
The branch unit is used for all types of branches including calls, returns and the vectoring to interrupt handlers.

The branch control unit internally generates the Committed Strand Mask (CSM) to determine which strands are committed and thus may generate valid branches.

7.2.1 Committed Strand Mask (CSM)

The CSM is a register that holds the set of strands that have been committed within the region. It is used to determine which branches that have been previously issued within the region are to be evaluated to determine if they are to be taken. The CSM is updated whenever a commit operation is performed. The branch from the lowest numbered strand is taken. All higher numbered strands are then squashed as they are not logically reached if an earlier branch has been taken.

The function used to update the CSM is illustrated below. A number of commit units may be supported. Each produces a valid output flag and the number of the strand that is being committed. The appropriate SEM bit for the strand is selected. If the strand is already disabled for whatever reason (due to a squash or abort etc.) then it is not included in the CSM. Otherwise a mask is produced for the bit representing the strand and this is included in the CSM. The CSM is passed to the branch resolution unit to select valid branches.



Note that a cumulative register is used for CSM rather than a combinatorial calculation to handle out of order commits. In normal circumstances strands are committed in their logical ascending order. However, in some circumstances the code generation tools may determine that greater code density can be achieved by committing the strands out of order with an appropriate guard to ensure that earlier strands have already been committed during a previous execution of the region. The CSM keeps a cumulative bit set of committed strands independently of the order in which they were committed.

7.2.2 Next Region Attributes (NRA)

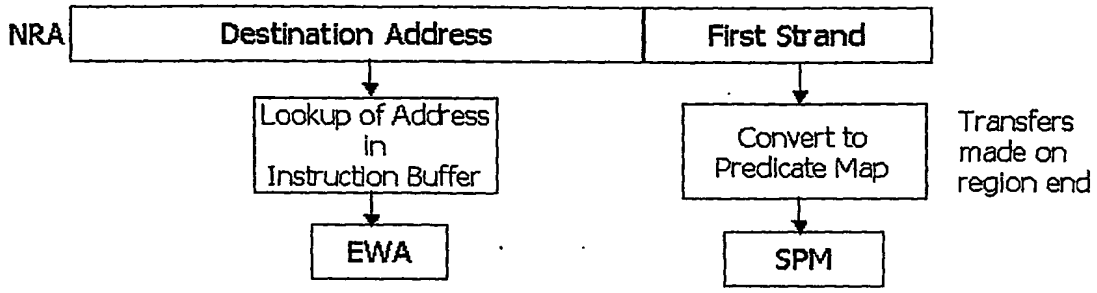
The NRA register contains the address of the next region that is to be executed. The branch resolution unit calculates the NRA as each strand is committed. The branch from the lowest numbered strand is selected as the destination. The branch does not occur until the end of the current region is reached.

The NRA may also be loaded with the address of an interrupt handler if an interrupt request is received. Such a request may be received at any point any is responded to immediately, even if the current region has not finished executing.

The NRA register has a shadow copy. This is used to preserve the NRA state if an interrupt occurs. Upon return from the interrupt the shadow copy of NRA is restored into the main register. This allows execution of the region to be correctly restored even if the destination region had been selected prior to the interrupt occurring.

The NRA consists of two fields. There is a full destination address that allows the specification of an address in the shadow memory in addition to the main memory. This is required for breakpoint branches. There is also a strand field for specifying the first strand that should be executed at the destination. This allows branches to side entries of a region.

When the end of the region is reached the NRA register is used to find the correct entry in the instruction buffer and to set the initial state of SPM (if an ordinary branch is being performed). The address of the region within the instruction buffer is loaded into EWA, from where execution of the region is commenced. This is illustrated in the diagram below:



7.2.3 Branch Issue

Branches operations may be issued to the branch unit. Branch operations only load the required destination information into the branch registers within the branch control unit. The actual branch is not performed until the end of the region is reached. Thus a multi-way branch is resolved at the end of the region.

Each strand is able to perform a single branch. A branch within a particular strand is unconditional. It is effectively performed at the end of the strand as the branch does not occur until the end of the region.

8 Region Creation

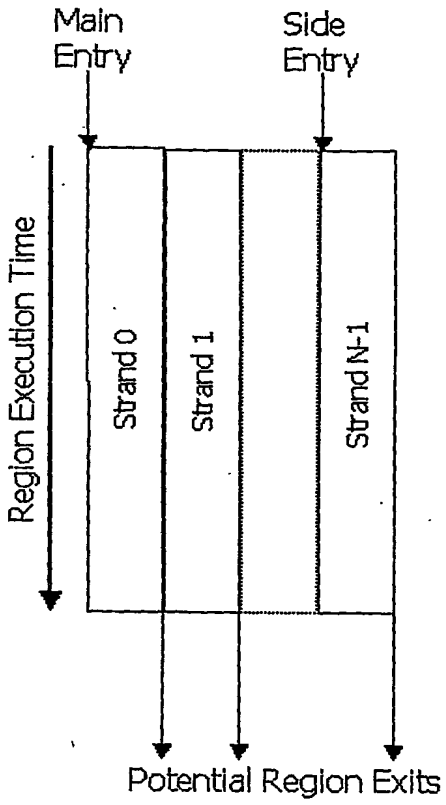
8.1 Region Definition

The code region is a fundamental building block for the CriticalBlue architecture, both during the code generation process and during code execution. Each function is composed of one or regions. A region is the basic unit of execution granularity. A region may only be entered via its first execution word and all execution words within it will be executed. A number of branches may be issued and they are resolved so that a single succession region is selected at the end of the region execution. If a cached instruction buffer is being used then a region is the unit that is loaded into the cache. A whole region is loaded as a single entity before its execution is commenced.

Each region may be composed of one or more strands. Strands are used to express internal control flow within a region. Strands have a fixed logical temporal ordering. Higher

numbered strands are logically later than lower numbered strands. The logical strand order is distinct from the physical execution order of operations within the region that is always performed through advancing addresses. The ability to express a logical ordering separately from a physical one allows speculative operations to be performed with recovery mechanisms using the strand numbering to generate the correct results.

The structure of a region is illustrated below:



A region has one or more entry points and one or more exits. Execution always continues until the last execution word in the region. The set of region exits are the set of regions that may be executed next. The main entry point for a region is always to strand 0. When a branch is made to a region a base strand number is specified. This is the first strand that is executed and lower strands are automatically squashed.

In addition to the main entry point a region may have a number of side entries. A side entry is simply an execution of the region that does not start with strand 0. Side entries are created when a previous strand performs a function call and the code after the call is part of the same region. A new strand is created at the point of the call and the return address is set to be a side entry to that strand. Side entries are also created when the region construction process encounters complex control flows in where there are control inflow edges to a strand that are outside of the region. In general the number of side entries within a region are minimised, especially for regions within performance critical functions.

8.2 Control Flow Analysis

Prior to region construction commences a full control flow analysis is performed of the function. Full control flow information is obtained for the function allowing the region analysis to follow outgoing edges from an instruction and incoming edges to it.

Each instruction in the function is examined. If the instruction is a branch then edge information is created an associated with the destination instruction. The fall through case when a conditional branch is not taken is considered to be a control flow edge in the graph, falling into the following instruction. By treating the fall through as an explicit edge this differentiates the control flow topology of an unconditional branch where there is no fall through.

8.3 Region Construction

A region is formed from a contiguous sequence of ascending instructions from the original code. The number of instructions that are included within a region is dependent upon many factors, especially the control flow topology of the included instructions. A region is not terminated by branches or calls and thus encompasses a much greater extent than a basic block.

The region is itself subdivided into one more individual strands. A strand represents an ascending sequence of contiguous instructions from the original code. Each successive strand is allocated a higher strand number. Thus in terms of relationship to the original code, the strand order is always the same as the original instruction order.

8.4 Strand Termination Conditions

8.4.1 Basic Block End

The strand is terminated when the end of a basic block is reached. A property of a basic block is that either all instructions within it are executed or none of them are. Thus the end of the basic block (and strand) is reached whenever a conditional branch instruction is reached. A basic block (and strand) is also terminated whenever an instruction is reached that may be branched to from elsewhere in the code. The control flow analysis generates a control flow graph that indicates which instructions have such inflow edges.

8.4.2 Conditional Instructions

The strand is terminated whenever a conditional instruction of any kind is reached. In most RISC architectures the only conditional instructions are branches (which end the basic block anyway). However, some architectures support conditional instructions such as conditional moves or include a conditional field in all instructions. The strand is terminated if the execution condition for an instruction differs from that for the whole strand. The CriticalBlue architecture does not support individual operations being marked as being conditional, they have to be mapped into different strands.

8.4.3 Call Site

The strand is terminated whenever a direct or indirect call is encountered. This is because the call itself does not occur until the region succession at the end of the region. Thus all code following the call must be placed into a separate strand. This and subsequent strands are executed on return from the function. In many cases a function call also causes the termination of the whole region.

8.4.4 Store Instruction

The strand may be terminated by a store instruction. This is done to improve the potential parallelism in the region. The CriticalBlue architecture supports speculative execution of load instructions by boosting them earlier than potentially aliased store instructions. However, this boosting can only be performed if the store and load are in different strands. Splitting strands at the point of a store allows this type of optimisation to be performed when the load operation is in the same basic block.

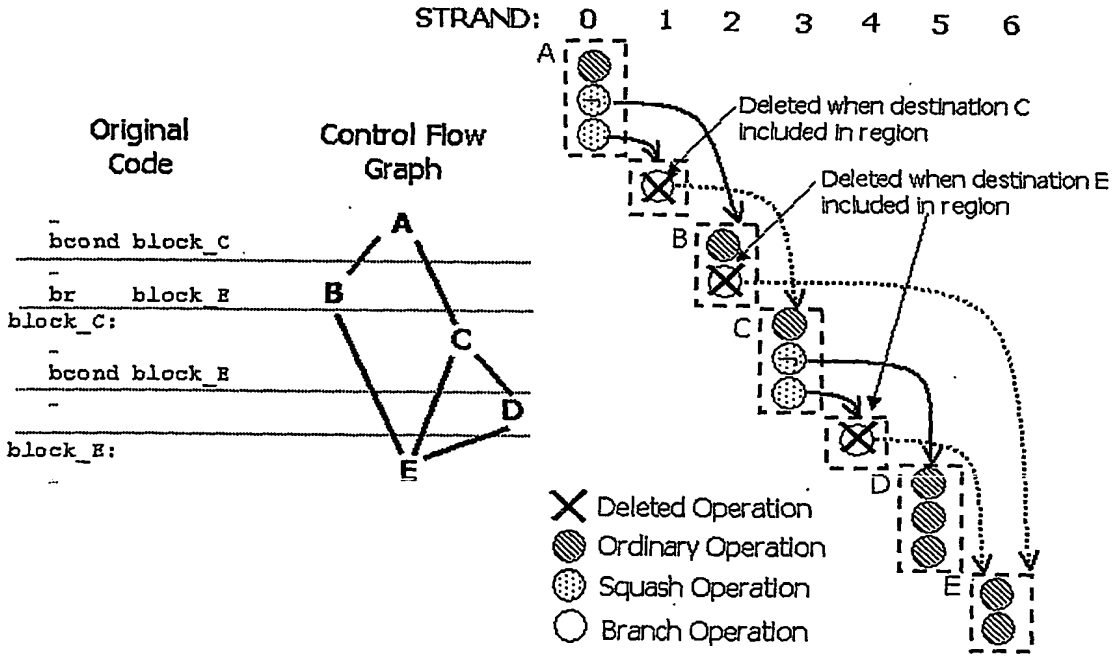
A store causes a strand split if there are subsequent loads in the same basic block that are potentially (but not definitely) aliased to the store. There is also a requirement that the store is the last such store in the basic block in order to prevent the basic block being split multiple times after each store instruction.

8.4.5 Excess Instructions

The strand is terminated if more than a certain number of instructions have been included within it. This prevents strands from becoming too long and subsequently creating regions that are larger than can be held in the instruction buffer. There is also a region termination condition that is triggered by an excess of instructions.

8.5 Conditional Handling

The following diagram illustrates how the strand creation process handles conditional code. A control flow graph is shown with basic blocks A to E. Block A is always entered at the start of the region. Blocks B and C are executed on a mutually exclusive basis and are formed from an IF-THEN-ELSE style construct. Block C also has a conditional block D formed from an IF-THEN style construct. Finally all control flow routes merge in basic block E.



Each basic block is allocated a new strand as illustrated on the right hand side of the diagram. A strand is composed of a number of individual operations, depending upon the instructions present in the original basic blocks. Basic block A terminates in a conditional branch that jumps over block B into block C. Strand 0 holds operations for block A and two squash operations are generated for the strand. The first squash controls the entry to strand B (the fall through case of the conditional branch). The second squash controls the execution of a specially generated strand 1. This strand holds the branch instruction to strand 3 (holding the code for block C).

As the code is first generated it is not known whether a conditional branch will be to code inside of the region or not. The extent of the region is determined as a dynamic process

so it is not possible to determine beforehand if an actual branch will be required or not. The region creation process assumes the worst case and generates branch operations. These are then deleted as soon as it is determined that the destination of the branch is within the region. For instance strand 1, which holds a branch to block C of code in strand 4 is deleted as soon as block C is encountered in the region. If the region was terminated before block C was reached then the branch would remain.

Strand 2, which contains code for block B, includes a branch operation. This is generated from the unconditional branch in the original code at the end of block B. Since the branch is conditional it can remain in the same strand as the rest of the operations for the block and it does not require a separate squash operation. When block E is reached (the destination of the branch) then the branch itself can be deleted.

When a branch is deleted and it is the only operation within a strand then the whole strand can be reclaimed. This is important since there is a limit of 16 strands that may be supported and if that limit is reached then the region must be terminated. Reclaiming strands prevents regions being falsely truncated by this limit.

Thus this process converts the control flow present in the original code into a sequence of strands with appropriate squash operations. Conditional blocks of code are converted into conditional strands, allowing much greater scheduling freedom. Only branches to destinations outside of the region remain as branches. This mechanism can convert arbitrary control flow into strand structures and can support the region being terminated at any point during the translation process.

8.6 Execution Density

8.6.1 Principle

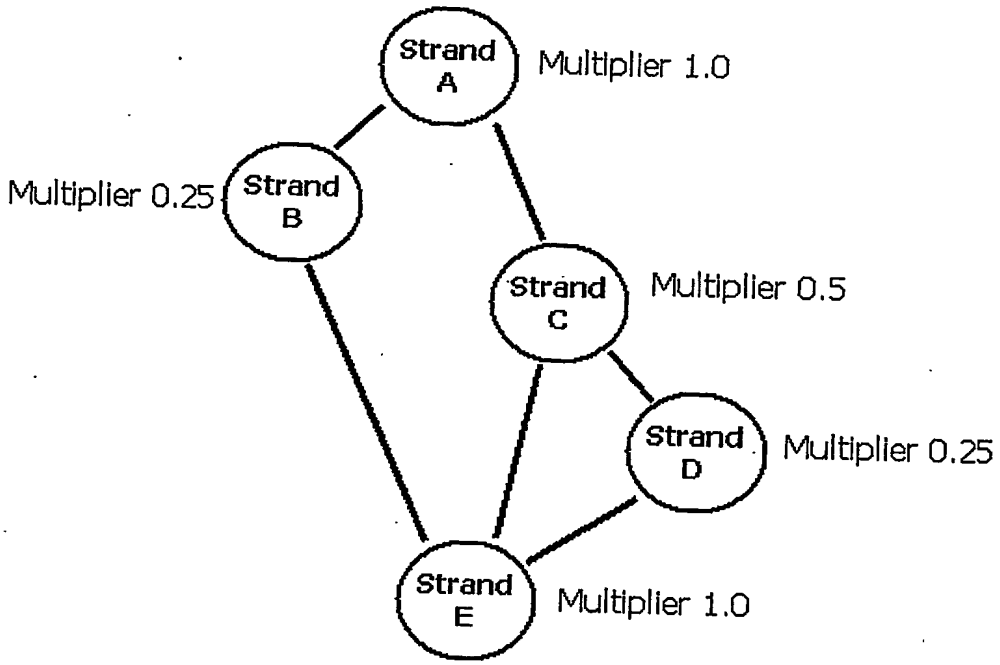
As a particular region is being constructed an execution density metric for the region is maintained. This gives an indication of the likely performance that will be obtained from the region. As a region expands it becomes more efficient in terms of code density. More code is packed into the region with the freedom to be scheduled throughout the execution cycles in the region. A better balance of operations is produced allowing operation slots in the execution word to be filled a higher proportion of the time. However, as the region expands it becomes less efficient in terms of performance. Each separate basic block that is included in the region as a new strand represents conditional code. If the logical control flow path does not go through the basic block then the operations issued as part of the strand will ultimately be squashed and their execution time will be wasted. In many circumstances those operations can be performed in parallel with unconditional operations so the overall performance impact is minimal. However, as more conditional strands are introduced the probability of executing the inner conditional code blocks becomes further reduced. A secondary effect is produced by calls and guarded strands. When a call is performed or a guarded strand is used the region must be re-executed. In the case of a call this is at the point of function return if code after the call is included in the region. In the case of guarded strand the region is re-executed if the strand is not the first to be executed in the region. These constructs also lower the effective performance of the region as redundant code has to be re-executed.

8.6.2 Density Calculation

An updated execution density is calculated for the region before the addition of each new block into the region. If the execution density falls below a certain threshold then the region is ended without adding the next block. The block becomes the first part of the following region.

In order to calculate the impact of a new block on the execution density a strand multiplier is calculated as each new strand is started in the region. If the candidate block initiates a new strand then this has a new strand multiplier associated with it. The strand multiplier essentially represents the probability that the strand will actually be executed. Profile information is not available to the region constructor so arbitrary assumptions must be made about the probability of branches in the code being taken.

The diagram below shows a control flow graph and associated multiplier values.



The initial strand in a region has a multiplier of 1.0. This indicates that the strand will definitely be executed on all entries to the region. A taken probability of 0.5 is assumed for all conditional branches. In the example strands B and C are mutually exclusive strands formed from an IF-THEN-ELSE style construct in the original code. Each path is given a multiplier of 0.5. Strand C has a further conditional code block that is given a 0.5 probability of being executed. Thus its multiplier is half that of its parent, giving a value of 0.25. Finally as the control flow points merge the multiplier is restored to 1.0 since it is definitely the case that strand E will be executed, regardless of the earlier control flow paths taken.

The strand multiplier is used to weight the count of instructions within a particular strand. This is cumulatively added to produce a cumulative weight for the whole region. In effect instructions from conditionally executed strands are partially discounted since there is only a probability that the instructions will actual perform useful work (i.e. that the strand will not be squashed). The execution density is simply calculated by dividing the total execution weight by the total number of instructions in the whole region.

The execution density is compared against a fixed factor to determine if the new block should be added to the region or not. This fixed factor varies depending upon whether the region is within a critical function or not. Critical functions are given a higher threshold than other functions. They must maintain a higher execution density because their execution performance is more critical. Ordinary control code is less crucial and more emphasis can be made on achieving good code density by allowing a lower minimum execution density.

8.6.3 Impact of Calls

Whenever a call is encountered a special adjustment is made to the execution density. This is to the account for the fact that the return from the function will re-execute the same region if the instructions after the call are included in the same region. Counting a call as more than one instruction makes the adjustment. A call is counted as the total number of previous instructions in the region multiplied by the strand multiplier for the strand in which the call is made. This adjustment is designed to take account of the fact that on return from the call all the previous instructions will have to be re-executed. If the call is made from a conditional strand then the impact is suitably reduced. The execution density is lowered since the total number of instructions forms the divisor for the calculation.

8.6.4 Impact of Side Entries

A side entry is a secondary entry point to a region other than through the very first strand or due to a call return. Side entries are detected when adding a new block to the region that has an inflow edge that does not emanate from an existing block within the region. This may be caused by complex control flow or the edge may be a back edge from a later instruction. Such a back edge normally indicates the start of a loop.

Whenever a side entry is reached the total cumulative instruction weight is halved. This has the effect of halving the execution density for the region up to the point of the side entry. In general side entries in regions are avoided as much as possible since if the side entry is taken all the previous instructions in the region are redundant and have to be squashed.

8.7 Loop Unrolling

In some circumstances loops within the code may be unrolled a number of times. This increases the code size but provides many benefits in terms of potential parallelism. The exit condition for the loop is duplicated each time the loop is unrolled so that it does not require a fixed number of iterations. Most unrolling optimisations require that the loop has a fixed number of iterations. Each unrolled invocation of the loop is mapped to one or more strands within a region. Thus operations from subsequent iterations can be executed speculatively in parallel with earlier iterations. Thus overall performance can be significantly improved.

For a loop to be unrolled it must be within a critical function. Loops within non-critical functions are never unrolled. There is a fixed limit to the number of times the loop will be unrolled. There are also limits on how big an unrollable loop may be in terms of the total number of instructions. The loop may contain internal control flow so that search loops with multiple exit conditions may be unrolled as well as simple linear code sequences.

The unrolling process is implemented by resetting the translation source address pointer back to the start of the loop. Thus the same block of loop code may be translated a number of times, producing a translated form in the CDFG that consists of a number of operation duplicates.

8.8 Region Termination Conditions

8.8.1 External Inflow Edge

The region may be terminated if the start of a block has an external inflow edge. An external inflow edge is one that emanates from outside of the current region. Inflow edges form within the region are acceptable as they represent control flow constructs that have been collapsed into strands. External inflow edges may result from complex control flow or be the entry points for loops in the code. If a critical function is being analysed then the region is always terminated as performance is considered to be a higher priority than

code density. If the function is not critical then the region may be extended across the inflow edge. However, the inflow edge will create a side entry that itself reduces the execution density and may cause the region to be terminated.

8.8.2 Backward Outflow Edge

A backward outflow edge causes the termination of the region as it represents the end of the loop and code after the loop should not be included in critical code. This region termination occurs for both critical and non-critical functions. If the region is part of a critical function then the loop may be unrolled when the back edge is reached.

8.8.3 Strands Exhausted

The region is terminated when the maximum number of strands is reached. The architecture supports up to 16 strands in a single region. An extra strand is always reserved to hold any branch operation associated with a block being translated. Additional strands may be reserved during the translation process to hold branches associated with conditions. If the branch destination is later shown to be within the same region then the reserved strand can be reclaimed.

8.8.4 Execution Density Too Low

A region is terminated if the execution density (including a new code block) is below a set threshold. The threshold is dependent on whether the function containing the region is critical or not. The execution density mechanism prevents regions being created that contain too much conditional code and are thus likely to provide inefficient performance.

8.8.5 Maximum Size

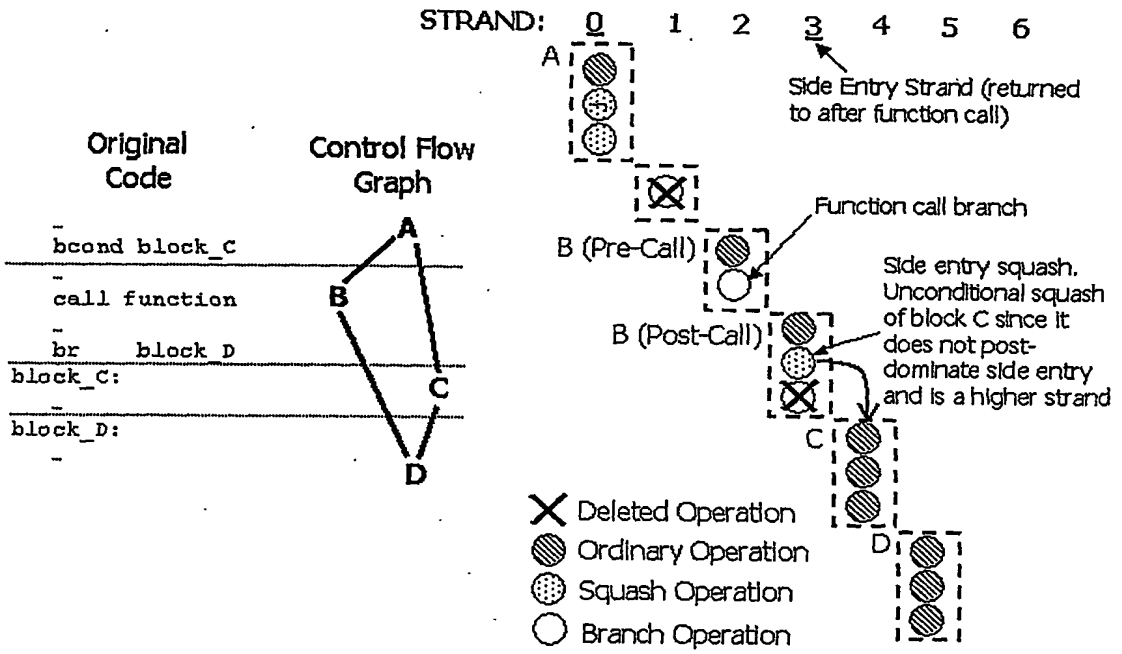
There are a maximum number of original instructions that can be included in a single region. This limit prevents regions becoming too large. This is a possibility if the original code is formed from a long linear sequence of code without conditional instructions or calls. The maximum size limit ensures that the region can be contained in the instruction buffer.

8.8.6 End of Code

Region termination occurs if the end of the code in the executable image is reached.

8.9 Side Entry Squashes

If a region has side entries then special unconditional squash operations have to be inserted into the side entry strand to ensure that only reachable strands are executed. This is illustrated in the following example:



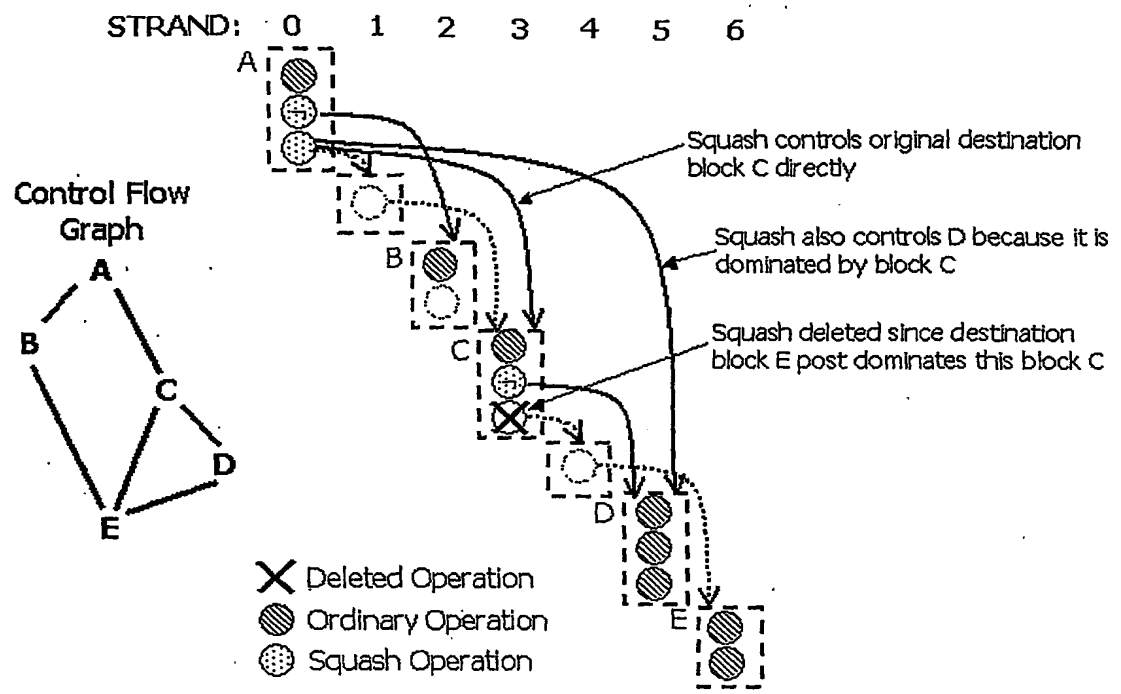
The original code forms an IF-THEN-ELSE construct. Depending on a condition evaluated in block A, either block B or block C will be executed. Two squash operations in block A disable the appropriate strand depending upon which path is taken. However, block B contains a call operation. If the path through B is taken then the whole region is executed and a branch is made to the called function. Block D is automatically squashed by the hardware as a branch from block B is taken so all higher numbered strands are squashed.

Block B is divided into two separate strands, 2 and 3. Strand 2 contains the translated code from before the call including the call itself. Strand 3 contains the code in block B after the return from the call. Strand 3 is a side entry since its address is used in an address link for return from the function. The return branch specifies strand 3 as the first strand to be executed in the region so strands 0 to 2 are automatically disabled and code form them is not repeated. However, strand 4 containing code from block C is enabled but this is should not be executed as the path through B has been taken. A special unconditional squash operation is inserted in strand 3 to disable block C. In general, for each side entry squashes are issued for all higher numbered strands that do not post-dominate the side entry.

8.10 Squash Resolution

When the construction of the region is completed the squashes within the region must be finally resolved. At this stage the strands are given their final numbering so that appropriate immediate strand values can be specified for the squash operations. During the construction process itself, strands may be reclaimed as they become empty if an unnecessary branch operation is deleted. Some squashes may be deleted at this stage if they are shown to be redundant.

The diagram below shows the set of operations and strand relationships for the example code used to illustrate conditional handling.



If a branch is deleted then the squash associated with that branch is modified to control the destination strand directly. For instance, the branch to block C has been deleted because block C is within the region. The squash within block A used to control that branch is made to control block C directly.

Some squashes may be used to control multiple strands. For instance, the squash for block C also controls block D with the same condition. This is because block D is dominated by block C. In other words, the code must pass through block C in order to reach block D. Block D is also controlled by a squash operation in block C but if block C is itself squashed by block A then that squash operation will never be executed. Each squash operation specifies a mask of strands to allow multiple squashes to be initiated by a single operation. If the necessary strands cannot be covered by a single squash operation then the squash resolution stage may insert additional squashes.

In general, a squash operation performed in strand x to control strand y is rewritten to also apply the same condition to all strands z, where z is dominated by x but does not post dominate x. Thus in the example block E is not included in the squashes of block A because all control flows pass through block E so its execution is unconditional. During the region construction process a matrix of strand dependencies is created, allowing domination and post-domination relationships to be determined.

All squashes to control strands that post-dominate the strand in which the squash is performed are deleted. This is the case with the second squash in block D to control block E. Whether the path through C is taken or not, block E is executed. Note that in the case of the squashes in block A, block C does not post–dominate block A (due to the unconditional branch at the end of block B) so the squash for it is retained. This rule allows appropriate code to be generated for differing control topologies of IF-THEN and IF-THEN-ELSE constructs.

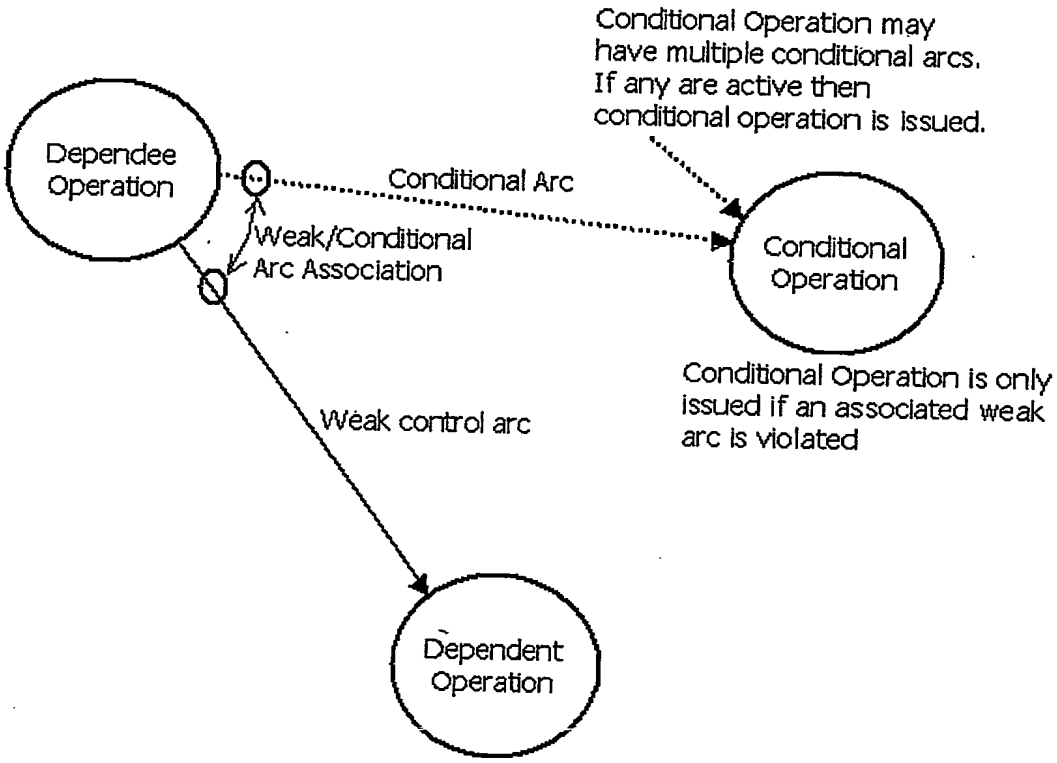
8.11 Weak and Conditional Dependence Arcs

8.11.1 Philosophy

Weak dependence arcs are used to provide hints to the scheduling algorithms about the ideal ordering of operations. However, unlike normal arcs the dependency rules can be broken and the operations issued out of order. This provides greater flexibility to the scheduler if it is attempting to improve code density by allowing greater scheduling freedom.

A weak dependence arc has an associated conditional arc. This arc is only activated if the ordering given by the weak dependence arc is broken. Conditional arcs are used to make the issue of certain operations, to compensate for the weak arc violation, conditional.

The diagram below illustrates the structure of weak and conditional arcs:



As shown the weak control arc has an associated conditional arc. This conditional arc is dependent upon a conditional operation that is only performed if the weak control arc dependency order is violated. If so then the conditional arc is denoted as activated. A conditional operation may be the dependee of a number of conditional arcs. If any of these arcs are activated then the conditional operation is issued. If a conditional operation is already activated then the scheduler does not need to attempt to take account of any other weak arcs associated with the same conditional operation.

8.11.2 Uses of Weak Arcs

There are a number of circumstances in which weak arcs are used:

- **Inter-Strand Register WAWs:** A weak arc is generated between the final write to a particular register in one strand and the first write to the same strand in a subsequent strand. There will be strong dependence arcs between writes to the

same register within the same strand. The weak arc ensures the ordering of register writes is the same as that in the original code. An associated conditional arc is generated between the earlier write operation and the guard operation for the later strand. Thus if the write dependency is violated then the later strand is guarded.

- **Inter-Strand Register RAWs:** A weak arc is generated between the final write to a particular register in one strand and all reads of the same register in subsequent strands. The weak arc ensures that the reads obtain the correct register values. An associated conditional arc is generated between the register write operation and the guard operation for the subsequent register reading strand. Thus if the write-read dependency is violated then the later strand is guarded.

- **Inter-Strand Memory WAWs:** A weak arc is generated between a write to a memory location in one strand and a write in a subsequent strand that is potentially or definitely aliased. The weak arc ensures the ordering of memory writes is the same as that in the original code. An associated conditional arc is generated between the earlier write operation and the guard operation for the later strand. Thus if the write dependency is violated then the later strand is guarded.

- **Inter-Strand Known Aliased Memory RAWs:** A weak arc is generated between a write to a particular memory location in one strand and all known aliased reads of the same memory location in subsequent strands. The weak arc ensures that the reads obtain the correct memory values. An associated conditional arc is generated between the write operation and the guard operation for the subsequent reading strand. Thus if the write-read dependency is violated then the later strand is guarded.

- **Inter-Strand Potentially Aliased Memory RAWs:** A weak arc is generated between a write to a particular memory location in one strand and all potentially aliased reads in subsequent strands. The weak arc ensures that the reads obtain the correct memory values. An associated conditional arc is generated between the write operation and a special chaz operation to guard the read. Thus the guarding chaz is only issued if the weak dependency arc is violated in the schedule (i.e. the read is issued earlier than the potentially aliased write). The chaz provides run time dynamic detection and recovery if the locations were actually aliased.

- **Strand Ordering:** A weak arc is generated between the squash operation in a strand and the phase barrier of a later strand that the squash guards. This prevents the committed phase of the later strand being started until the squash condition has been evaluated. An associated conditional arc is generated between the squash operation and the guard operation for the later strand. Thus if the dependency is violated then the later strand is guarded.

- **Method Ordering:** A weak arc is generated between an operation in a strand and a preceding operation in a previous strand that is a member of the same dependence set. This enforces an ordering between the operations if they can potentially both be issued during the same region execution.

8.12 Memory Dependence Analysis

After a memory access operation is issued into the CDFG, memory dependence analysis must be performed. This involves checking to determine if the access could be aliased with a memory access issued earlier in the CDFG. The accesses are aliased if they may potentially access the same memory location. If so, and one operation is a load and the

other a store, then their ordering cannot be changed in the schedule as that could impact the results generated by the program. Dependence arcs are created between potentially aliased operations to ensure the correct ordering is maintained in the final schedule.

8.12.1 Alias Checking

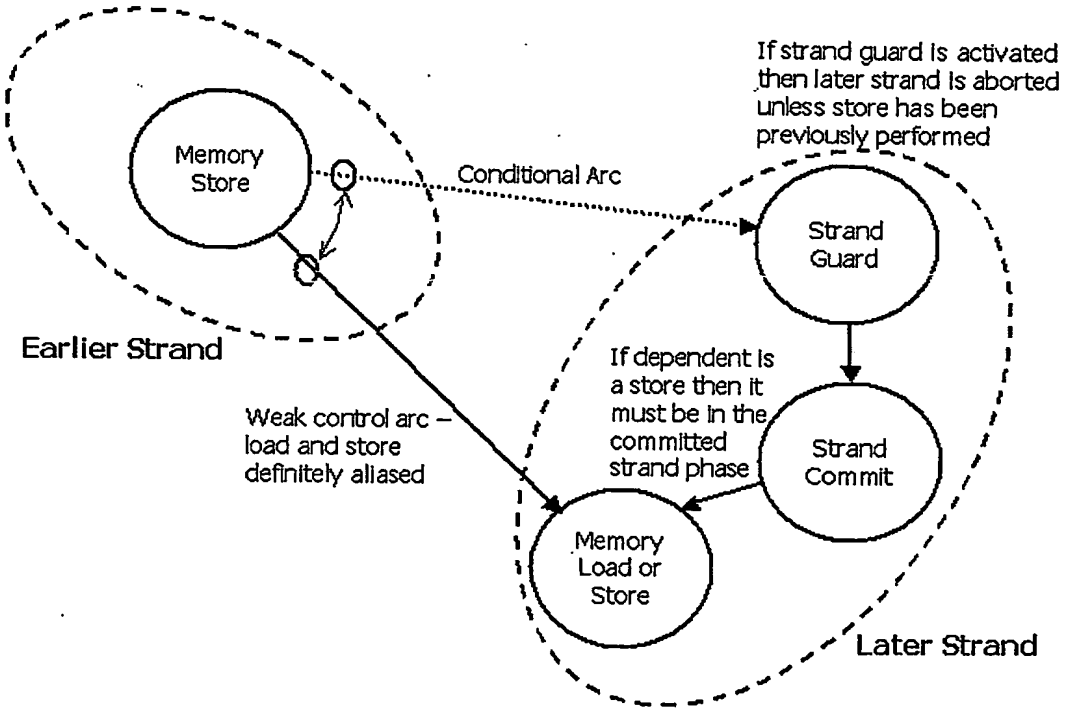
Alias checks are made with all previous stores issued in the CDFG that are reachable from the strand of the new operation. If the previous store is from a strand that is on a mutually exclusive control flow path in the region then no dependence arc needs to be generated. For instance, a region may include an IF-THEN-ELSE control flow structure. Memory accesses in the ELSE part of the construct will not have dependency arcs generated with operations from the THEN part as the strand squashing will ensure that both paths are not executed. Thus memory access operations from the ELSE part can be issued before potentially aliased memory access operations from the THEN part.

If a previous store is aliased then the type of dependency arc generated will depend upon a number of factors. If the previous store is within the same strand then an ordinary control arc is generated between the two nodes to ensure they maintain the same order. If the nodes are in different strands then a weak control arc may be generated. An associated conditional arc will also be generated. This mechanism allows memory accesses to be performed out of order if appropriate compensation measures are put in place. This is described in more detail in the following sections.

8.12.2 Known Aliased Dependence

In this case there is a known aliased dependency between a store and a load or any potential alias between two stores. The two memory operations will be in different strands (or else an ordinary control flow arc is used).

The dependencies are illustrated in the following diagram:



A weak control arc is generated between the two memory accesses. A separate conditional arc is generated between the earlier store and the guard operation of the later

strand containing the dependent instruction. These two arcs are associated with one another. If the weak control arc is violated (i.e. the later operation is issued before the earlier store) then the conditional arc is activated. This means that the strand guard operation is issued. If there are no dependency violations then the guard operation is not issued in the schedule. If the later operation is a store then it will be a dependent of the strand commit operation since stores must be issued in the committed phase of the strand.

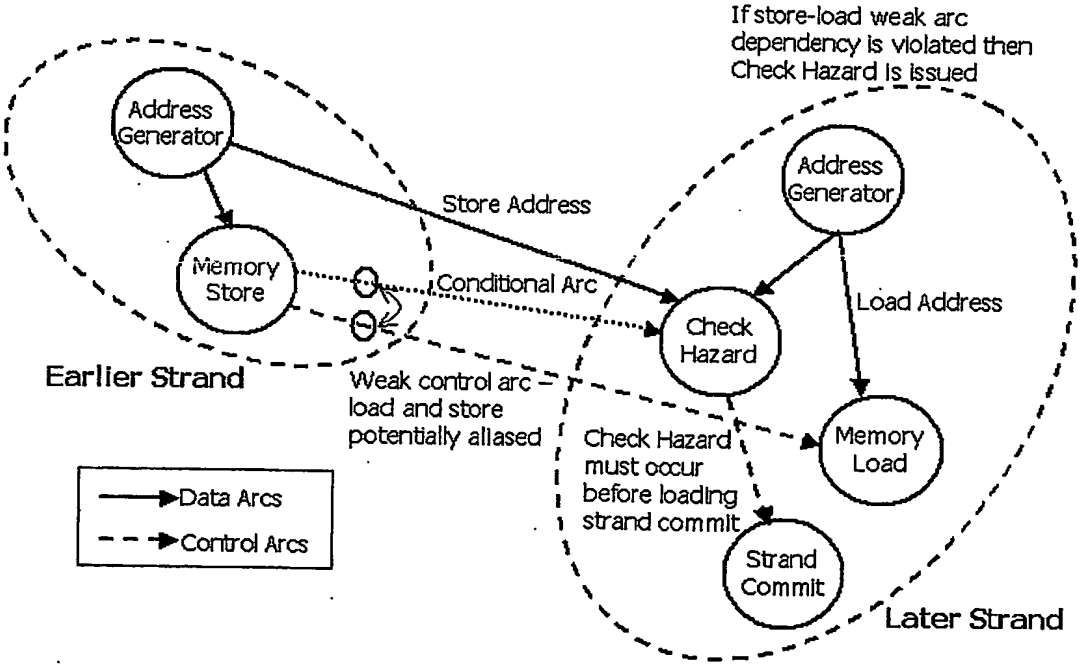
If the weak arc dependency is violated then the issue of the guard operation ensures that correct results are still generated. A strand guard causes the strand to be aborted unless it is the first strand being executed from the region. If the store (from an earlier strand) is being executed then the later strand is aborted. This is because it will have issued a load illegally before the store. This causes the later strand to be re-executed. This time the guard will not cause an abort and the load can read the data written by the earlier store.

Clearly an abort and re-execution of the strand is not efficient in performance terms. In performance critical functions, strong dependency arcs are generated to ensure that the memory access order is not changed. Conditional aborts are used in other functions where the greater scheduling freedom allows improved code density. The scheduler attempts to recognize the weak dependency if possible.

8.12.3 Potentially Aliased Dependence

In this case there is a potential dependency between an earlier store and a later load operation. In general it is unlikely that the two accesses will actually be aliased. In most cases issuing the load earlier than the store will still produce the correct results. However, the architecture must detect the cases in which they are aliased and provide an appropriate compensation mechanism.

The following diagram shows an example of potentially aliased access handling:



A weak control arc is generated between the store and the later load. A special check hazard operation is issued in the later strand holding the load. A conditional arc is created

from the store to the check hazard operation. Thus if the load is issued before the store in the schedule then the check hazard operation is issued.

The check hazard operation requires the addresses generated for the store and load operations. The check hazard operation obtains those from the same operations that generate them for the store and load. Although the check hazard is issued in the subsequent strand it is able to use the address calculated for the store in the earlier strand. Data flow between strands in this manner is not normally possible. However, either the store or load strand can disable a check hazard. If the store address is not valid then the operation is not performed.

The check hazard also depends on the commit operation for the loading strand. This is because the check hazard must be issued in the speculative phase of the loading strand as it has the potential to abort the strand. A check is made to ensure that the address generation for the load is not a dependee of any operations that must be issued in the committed phase of the loading strand. This would not be legal as it would make the graph acyclic, as the check hazard must be issued in the speculative phase.

The check hazard operation simply compares the load and store addresses. If they are not equal then the operation has no effect. Thus if a load is issued earlier than a potentially aliased store but the addresses are not actually aliased at run time then execution can continue normally. If the addresses are identical then the strand to which the check hazard belongs (the load strand) is aborted. The later store can then be performed. The region is then re-executed and then check hazard will not be performed as the store address will be generated from a disabled strand. On the re-execution the load obtains the correct data from the store performed previously.